Humboldt-Universität zu Berlin

HABILITATION

Formal Semantics for SDL

Definition and Implementation

zur Erlangung der Lehrbefähigung

für das Fach Informatik

vorgelegt dem Rat

der Mathematisch-Naturwissenschaftlichen Fakultät II

der Humboldt-Universität zu Berlin

von Dr. Andreas Prinz
geboren am 12. März 1963

Dekan: Prof. Dr. Bodo Krause

Präsident der Humboldt-Universität zu Berlin: Prof. Dr. Dr. h.c. Hans Meyer

Gutachter: 1. Prof. Dr. Joachim Fischer

2. Prof. Dr. Egon Börger

3. Prof. Dr. Klaus Bothe

eingereicht: 22. Juni 2000
Datum der Verteidigung: 23. Mai 2001

FOREWORD

There is nothing as practical as a good theory.
Helmut Thiele

Formal languages play an important role in mathematics and in computer science. In contrast to natural languages, formal languages have two important properties. First, it is always clear whether or not a particular sentence (also called a "formula" in mathematics, or a "program" or "specification" in computer science) belongs to the language. This property is usually called the *syntax* of the language. Second, the meaning of the valid sentences is clear. This is called the *semantics* of the language.

Nowadays the description of a specification language contains a formal syntax and an informal semantics description. However, it is important to state the semantics formally as well in order to have a formal language.

This book will present a complete description of a sample language: starting from the informal description of the language and progressing to a formal semantics. Moreover, this book will show how the formal syntax and semantics are implemented.

There are some prerequisites which readers should have in order to get the maximum benefit from this book. First, in the area of software technology, you should know a fair amount about programming. It is assumed that you can read C/C++ program fragments and have basic knowledge about what compilers and interpreters do. In the area of formality, you are expected to have a basic understanding of formal methods, logic or similar areas of mathematics. A fair understanding of set theory and first order logic is required, as these fields underlying the ASM framework are not explained within the book.

ACKNOWLEDGEMENTS

It is always an adventure to undertake a project like the one documented here. In this case, it would not have been successful without the active support of many people. I am very grateful to all of them.
Special thanks go to the following people.

- My wife Lisanne and our children Felix, Flora and Frieder, as well as our parents for releasing me to do the writing, and for tolerating that I was so absorbed in the work.

- Prof. Joachim Fischer for continuously supporting my work, for promoting the formal SDL semantics development and for providing good working conditions.

- Dr. Michael Weber for his generous technical and financial support in the company DResearch and for his active interest in the work.

- Prof. Reinhard Gotzhein, Dr. Uwe Glässer and Robert Eschbach for contributing their expert knowledge and for many fruitful discussions and meetings concerned with the SDL semantics.

- Wang Ying and her colleagues for their active work on the SDL static semantics conditions.

- My colleagues at the Humboldt University for many important discussions about SDL and its semantics.

- The SDL standardisation group for carefully designing the informal SDL semantics and for many valuable meetings about the language SDL.

- Many test readers for their comments and advice on earlier drafts of the text.

TABLE OF CONTENTS

Part 1: INTRODUCTION

This book is about Software Technology.
This book is about Language Semantics.
This book is about Formal Methods.

Formal languages play an important role in mathematics and in computer science. In contrast to natural languages, formal languages have two important properties. First, it is always clear whether or not a particular sentence (also called a "formula" in mathematics, or a "program" or "specification" in computer science) belongs to the language. This property is usually called the *syntax* of the language. Second, the meaning of the valid sentences is clear. This is called the *semantics* of the language.
Nowadays the description of a specification language contains a formal syntax and an informal semantics description. However, it is important to state the semantics formally as well in order to have a formal language.
This book attempts to describe the formal semantics of the standardised specification language SDL (Specification and Description Language). Because SDL is a very large language, a restricted language RSDL (Restricted SDL) was selected in order to present the formal definition concepts of SDL. The RSDL subset of SDL was chosen such that it is small enough to be completely covered within this book but, still has the same complexity as SDL. This makes it possible to use the same method as described here for the formal SDL semantics as well. More information is provided about the SDL-2000 semantics project in Section 6.5.
Two major topics are covered within this book, namely the RSDL formal semantics definition and its implementation. This results in the following overall layout.

- Part 1: Opening
  The first part is intended to provide an overview of the book, and to explain some basic conventions that are used throughout.
- Part 2: Basic Information
  In the second part all the ingredients of the language semantics are presented, namely the language ASM and the tools used.
- Part 3: Language Description
  The language RSDL is described with its syntax and informal semantics.
- Part 4: Formal Description
  Here the language ASM is used to give a formal semantics of RSDL.
- Part 5: Implementation Aspects
  This part shows how the formal description is actually implemented using the tools presented in Part 2.
- Part 6: Closing
  The last part is devoted to various surrounding and background information.

Please see Section 1.6 for a more detailed description of the implementation and its connection to the remaining parts of the book.

1.1 Motivation

There is a need to provide formal definitions of specification languages. This need comes from a desire to have better ways to verify properties of specifications, as well as to provide better means to check the correctness of specification language tools.
The idea is simple: If the formal semantics of a specification language is given without referencing any implementation details, one can then check a concrete implementation against this description for correctness. The implementation details can be chosen by implementations and are not prescribed by the specification. If a formal definition of the specification language semantics is provided, then properties of the system can be derived without even implementing it.
In the following, we survey some areas where formal methods are beneficial.

1.1.1 Formal Grammars

Formal grammars are used to formally define syntactical structures. A *grammar* consists of four parts: a set of *terminal* symbols T, a set of *non-terminal* symbols N, a *start symbol* $s \in N$, and a set of *grammar rules* R. Each grammar rule has a left-hand side LHS and a right-hand side RHS. Both LHS and RHS are sequences of terminals and/or non-terminals, i.e. $LHS \in (T \cup N)^*$, $RHS \in (T \cup N)^*$. The grammar rules recursively define a set of valid character sequences. The Backus-Naur Form (BNF) is an example of a formal grammar notation,

a rule is written here as LHS ::= RHS. The BNF was originally developed by Backus and Naur for the prescription of the syntax of the ALGOL 60 programming language.
Please find below the grammar for simple arithmetic expressions.
T={“0”, “1”, “+”, “-”, “*”}, N={expr}, s=expr,
R={<expr, “0”>, <expr, “1”>, <expr, expr “+” expr>, <expr, expr “*” expr>, <expr, “-” expr> }.
The set R could be expressed by the following BNF rule.
expr ::= “0” | “1” | expr “+” expr | expr “*” expr | “-” expr

The success of formal grammars led to the current situation in which specification languages have a formally defined syntax, most often based on some variant of BNF. The language grammar is used by tool developers to build tools and by language users to understand language constructs.

1.1.2 Formal Semantics

The definition of the semantics of some specification languages (e.g. UML) is given in ordinary prose. In order to define the syntax of a language construct, a set of grammar rules in BNF or another grammar formalism is provided. This formal syntax is accompanied by a few paragraphs and (hopefully) a number of examples to define the semantics. Unfortunately, the prose is sometimes ambiguous leading to different interpretations of the semantics of a language construct. This may affect both users of the language and tool developers. First, a language user may misunderstand the specification, and tool developers may also implement a specification construct in a different manner from other developers of tools for the same specification language. Hence, as with syntax, methods are required to provide a precise, readable and concise definition of the semantics of a specification language.

1.1.3 Language Design

When the semantics of a specification language is defined formally, some interesting questions can be asked:
- What relations exist between language constructs?
- Can some language constructs be derived from others?
- Can the combined use of language constructs cause problems?

Although many of these questions were asked without a formal semantics, it is now possible to examine them by formal means. Moreover, the formalisation process itself will uncover omissions and inconsistencies in the language definition.

1.1.4 Verifying the Specification

Formal semantics can be used to mathematically verify properties of the specification. In this way every possible behaviour of the specification can be covered, not only those that are considered during tests or during use. The following questions can be addressed.
- Does the specification contain a deadlock?
- Does the specification contain an infinite loop (livelock)?
- Will the specified algorithm always terminate?
- How long will it take to compute the result?

To state such questions formally, it is necessary to have an appropriate mathematical description of the specification language (and thus of the individual specification).
Please note that some of the questions above are in fact not decidable, i.e. there is no way to find an algorithm to check such a property automatically.

1.1.5 Type Safety

A formal definition of the semantics of a language permits a definition of typing. A correctly typed specification does in fact make statements about language constructs or data such as: *In this specification, no data flows into a place that is not capable of holding it.*
Typing is a property that is statically decidable, i.e. decidable without interpreting the specification. However, typing states constraints that are satisfied by the specification when interpreted. Most modern specification languages introduce polymorphic typing. By means of a formalisation it is possible to prove the correctness of the typing rules, i.e. that static type correctness implies dynamic type correctness.

1.2 Semantics Definition Styles

The problem of language semantics definition has been a research topic for a considerable period. However, unlike the area of syntactical definition, satisfactory solutions have been rare. Although semantics definitions for mathematical languages are well-known, defining the semantics of specification languages has turned out to be more difficult. Specification languages are often larger than mathematical languages; they have many special cases; and they have dynamic semantics, i.e. the meaning of a construct depends on the state of the whole system.

Many different methods of formal definition have been developed, and these may be divided into three general classes:

- operational techniques,
- denotational or functional techniques, and
- axiomatic techniques.

Unfortunately, no single method is appropriate for both users and tool developers. Sometimes a semantics does not purely follow one of the above styles, but is a mixture of them. In such cases it is often valuable to identify the parts that are covered by each style in order to gain better overview of the semantics. In the following, the different styles are explained in more depth, using as an example the simplified arithmetic expressions defined in the section on formal grammar above.

1.2.1 Axiomatic Semantics

Axiomatic techniques for specification language semantics were derived from mathematical logic, logical equations and model theory, motivated by a desire to perform program correctness proofs. The entities of the language and their relations to each other are identified. For our example, we have the entities expr, "0", "1", "+", "-", "*" as indicated by the following declarations.

domain	expr	; expr is the only domain set
constants	"0", "1"	; 0 and 1 denote elements of the domain expr
functions	"+": expr × expr → expr	; this defines the signature, i.e. type of the addition function
	"*": expr × expr → expr	; this defines the signature of the multiplication function
	"-": expr → expr	; this defines the signature of the minus sign

Their relations are captured by the following axioms.

axioms for all x,y,z: expr

1. $x \neq 1+x$
2. $0+x = x$
3. $x+y = y+x$
4. $x+(y+z) = (x+y)+z$
5. $1*x = x$
6. $x*y = y*x$
7. $x*(y*z) = (x*y)*z$
8. $x*(y+z) = (x*y) + (x*z)$
9. $x+(-x) = 0$
10. $-(x+y) = -x + -y$
11. $-(x*y) = -x * y$

From the axioms above together with a suitable substitution property we can for example derive that $0*e=0$ holds for any expression e as follows.

$0*e$	=	$(1+(-1))*e$	(by 9)
	=	$e*(1+(-1))$	(by 6)
	=	$(e*1) + (e*(-1))$	(by 8)
	=	$(e*1) + ((-1)*e)$	(by 6)
	=	$(e*1) + (-(1*e))$	(by 11)
	=	$(e*1) + (-(e*1))$	(by 6)
	=	0	(by 9)

We can also derive that $1 \neq 0$, because $1 = 1+0 \neq 0$.

The benefits of the axiomatic method are as follows:
- It provides a very abstract semantics definition.
- There is no impact on an implementation.
- Mathematical methods (proofs, model checking) can easily be used.
- The axioms are concise and understandable.

But there are also problems:
- Little or no guidance is provided to tool developers.
- The description tends to be very large when many basic constructs are considered.
- It is very complex for real languages.
- It is difficult to formalise operations and states.
- There is no easy overview of the implications of the definition.

The sample language illustrates the implications problem. The intention of the definitions above was to provide a description of the integers. We are therefore tempted to conclude $0 \neq 1+1$. However, this is not implied. The equation $1+1=0$ would be perfectly valid if the domain expr contained only the elements 0 and 1. All axioms hold in this case.
To avoid writing too many axioms, axiomatic semantics is often restricted to what are called initial models. This means that we only want to consider the most general models matching the definition. This is often stated in the form of two conditions: *no junk* and *no confusion*. 'No junk' means that we do not want to include elements in a domain that are not really implied by the language definition, e.g. we do not want to have π in the domain expr. 'No confusion' means that we do not want to regard elements as being the same unless explicitly stated. In the above example, this would mean that we have an implicit axiom $1+1 \neq 0$.

1.2.2 Denotational Semantics

The basic idea of denotational semantics is to give a denotation to every element of the language. This means the syntactical expressions of the language are mapped to a well-known domain. For the sample language, we define a mapping of the language entities to the integers. The *denotation function* is often called [_] as in the definitions below, where *x* and *y* are variables of the domain Integer.

[expr]	=	Integer
["0"]	=	0
["1"]	=	1
[*x* "+" *y*]	=	[*x*] + [*y*]
[*x* "*" *y*]	=	[*x*] * [*y*]
["–" *x*]	=	– [*x*]

Please note that there is always an implicit axiomatic semantics hidden in this approach: in the example, this is the semantics of integer, which is considered to be predefined in ordinary mathematics. The general concept of denotational semantics is to map the unknown language to a known language. This basic known language should itself have a formal semantics given with one of the three styles.

The benefits of the denotational semantics are as follows.
- It resembles the syntax structure.
- It builds on known domains.
- The semantics description is fairly abstract.

However, there are also problems:
- It provides but little guidance to tool developers.
- It is usually too complex for users.
- For complex languages, the target domains are not readily available.
- There are again difficulties in expressing states and operations.

The denotational approach works better where the mapping is easier. In the example, we see that the mapping is one-to-one and therefore easy to read. However, it is much more difficult to map a real language, such as C for example, to basic mathematical domains. In such a mapping, various auxiliary functions must be introduced and one element of the source language is mapped to many elements of the target language. Therefore, a common approach is to first define a specialised target language axiomatically and then to give a denotational semantics based on this special language.

1.2.3 Operational Semantics

The operational approach is the most concrete one, and it is very near to implementation. The idea is to interpret the specification in an abstract interpreter. The abstract interpreter is a program of an abstract computer (e.g. a Turing Machine). The operational semantics of the sample language in the previous sections is given below using an abstract Pascal style for the interpreter program:

```
procedure compute(e: expr) returns integer is
  case e of
    "1": return 1;
    "0": return 0;
    "+": return compute(e.first) + compute(e.second)
    "*": return compute(e.first) * compute(e.second)
    "–": return – compute(e.first)
  endcase
endprocedure
```

The operational method also uses a predefined semantics, namely the semantics of the abstract computer. In fact, this abstract computer semantics may in turn be given using any of the three semantics definition styles. However, the semantics of the abstract computer need not be complete, because only one program—namely the interpreter program—is interpreted. It suffices for the machine to handle this single program. Using the operational method one could even define the semantics of a language in (a restricted version of) itself using some kind of bootstrapping.

The operational method has the following benefits:
- It provides a good formalisation of implementation.
- It is easily understandable for tool developers.
- It is well suited for state-based languages.

Again, we have some problems:
- Operational descriptions tend to be very detailed.
- It is very difficult to derive formal proofs from an operational semantics.
- The operational approach needs an underlying semantics of an abstract computer.

A similar remark as for denotational semantics is appropriate here: the operational approach is easier to understand when the underlying abstract computer matches the paradigm of the source language. Therefore it is quite common to build first a special abstract computer which is tailored to the source language to provide an easy interpretation.

1.3 Notational Conventions

Before we look into the RSDL semantics definition, this section explains the notational conventions used throughout the book.

1.3.1 Typography

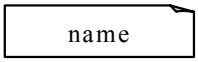
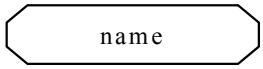
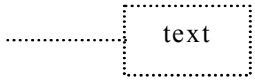
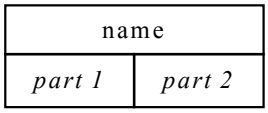
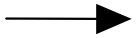
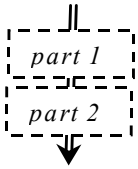
In order to make the book more readable, several special typefaces are used to distinguish the different kinds of names in different contexts. Please find in the table below a list of typefaces used together with their meaning.

Word	Keyword
`Word`	Code
<word>	language definition: concrete syntax name
Word	language definition: abstract syntax name
WORD	semantics definition: domain name
Word	semantics definition: function or variable name
WORD	semantics definition: program or macro name

A more in-depth description of the typefaces used for the semantics definition can be found in Section 2.1.1.1, Section 2.1.1.6 and Section 2.1.2.3.

1.3.2 Elements in Figures

All the figures are designed to be understandable without additional explanation. However, as a further aid to understanding, the various pictorial elements all have a fixed meaning for all figures. The following table shows the symbols used with their respective meaning.

	data (text, file, etc.)
	processing (step, unit, etc.) possibly nested
	data flow
	comment box, optionally associated with an element
	informal partitioning line
	concept, notation, language (with parts)
	is defined using, is an extension of
	mapping symbol and parts of the mapping; if only one part is given, it denotes the name of the mapping.

1.4 Formal Semantics for RSDL: Overview

This section provides a top-level overview of the semantics of RSDL. The same methodology is applied for the formal SDL-2000 semantics [27]. Both the informal and the formal language descriptions consist of a syntax (static description) and a semantics (dynamic description). The static description explains which programs are correct in the language. The dynamic description explains the behaviour of correct programs.

1.4.1 Static Description

The static formal language definition consists of the following parts as shown in Figure 1:

- a concrete syntax,
- a set of well-formedness conditions,
- a set of transformation rules, and
- an abstract syntax as a basis for the dynamic semantics.

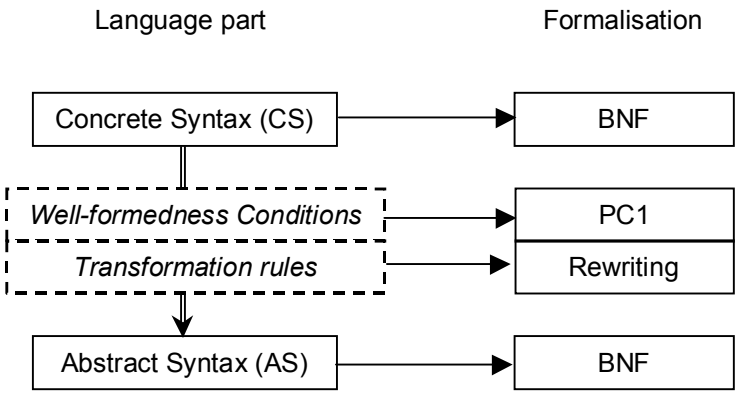


Figure 1: Static Part of RSDL

The *syntax* defines the set of syntactically correct RSDL specifications. For RSDL we make a distinction between a concrete and an abstract syntax. These are defined formally using BNF with some extensions for capturing disambiguation. The abstract syntax is obtained from the concrete syntax by removing irrelevant

details such as separators and lexical rules. Moreover, shorthand notations are not represented within the abstract syntax. They are replaced by their corresponding basic constructs (see also transformations below).

The *well-formedness* conditions define additional conditions that must be satisfied by a well-formed RSDL specification, and which can be checked without interpreting an instance. An RSDL specification is *valid* if and only if it satisfies the syntactic rules and the static conditions of RSDL. In fact, the static conditions refer to the syntax, but they have not been stated in the concrete syntax because they are not expressible in a context-free grammar.
There are basically five kinds of well-formedness conditions:

1. Scope/visibility rules: the definition of an entity introduces an identifier that may be used as the reference to the entity. Only visible identifiers may be used. The scope/visibility rules are applied to determine whether the corresponding definition of an identifier is visible or not.
2. Disambiguation rules: Sometimes a name might refer to several identifiers. Rules are applied to find out the correct one.
3. Data type consistency rules: these rules guarantee that at interpretation time no operation is applied to operands that do not match their argument types. More specifically, the data type of an actual parameter must be compatible with that of the corresponding formal parameter; the data type of an expression must be compatible with that of the variable to which the expression is assigned.
4. Special rules: there are some rules which are applicable to specific entities. For example, there must be local blocks or a graph within a block.
5. Plain syntax rules: There are some rules which refer to the correctness of plain syntax constructs that do not get transformed into the abstract syntax, such as the rule that the name at the beginning and at the end of a definition match.

The static semantics of RSDL is defined in terms of first-order predicate calculus (PC1).

Furthermore, some language constructs appearing in the concrete syntax are replaced by other language elements in the concrete syntax using *transformation rules* in order to keep the set of semantic core concepts small. These transformations are given in the language description. Formally they are represented as rewrite rules. A single transformation is realised by the application of a rewrite rule to the concrete specification, which essentially means to replace parts of the specification by new parts as defined by the rule. Moreover, the RSDL informal semantics defines transformation steps that are given by sets of rewrite rules. Each of those steps defines how to handle one special class of shorthand notations. The result of one step is used as input for the next step.
It is important here to correctly identify the core concepts corresponding to the intuitions behind the language design in order to facilitate easy transformation. If there are too many concepts, giving a semantics is unnecessarily complicated. If there are too few or the wrong concepts, the transformations tend to be very complex and their meaning is no longer easily understood.

1.4.2 Dynamic Description

The *dynamic description* is given only to RSDL specifications that are valid. In addition, the transformations must have been applied. The dynamic semantics defines the behaviour associated with a specification.
The dynamic semantics starts with the result of the static description, i.e. with the *abstract syntax*. In order to better show the structure of the dynamic description, we identify three parts of the abstract syntax, namely *structure*, *behaviour* and *data*. The dynamic semantics is based on a mathematical theory called Abstract State Machines (ASM). All parts between the abstract syntax and ASM belong to the dynamic description. There are four such parts, as can be seen in Figure 2.

- A *Special Abstract Machine (SAM)* which is defined using ASM. For better match with the abstract syntax, we identify three parts of the SAM, namely basic features to express *structural* properties, *connections* (RSDL channels and other RSDL connections) and *behaviour primitives*, in a way the abstract machine instructions.
- An *initialisation* which is necessary to handle static structural properties of the specification. The initialisation does a recursive unfolding of all the static objects of the specification. In fact, the same process will be initiated at interpretation time too, when new RSDL agents are created. From this point of view, the initialisation is merely the instantiation of the outermost RSDL agent.
- A *compilation* function that maps behaviour representations into the SAM primitives. This function amounts to an abstract compiler taking the abstract syntax of the specification as input and transforming it into the abstract machine instructions (see SAM).
- A *data semantics*, which is separated from the rest of the semantics by a data interface. The use of an interface is intentional at this place: it will allow the data model to be exchanged if for some domain another data model is more appropriate than the built-in model. Moreover, the built-in model can also be changed without affecting the rest of the semantics.

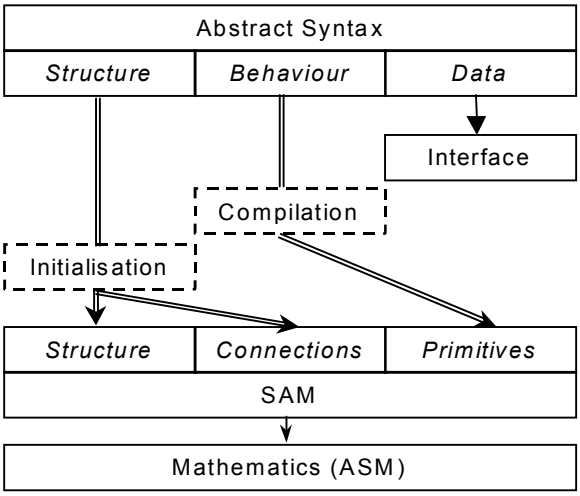


Figure 2: Structure of the Dynamic Semantics

The formal semantics is defined starting from the abstract syntax of RSDL. From this abstract syntax, a behaviour model is derived. The approach chosen here has the characteristics of all three basic semantics definition styles. The SAM is expressed in terms of ASM, and like ASM has a mathematical definition (an axiomatic semantics). The compilation defines an abstract compiler mapping the behaviour parts of RSDL to abstract code (denotational semantics). Finally, the initialisation describes an interpretation of the abstract syntax tree to build the initial system structure (operational semantics).

The *dynamic semantics* associates a particular multi-agent real-time ASM with each RSDL specification. Intuitively, an ASM consists of a set of autonomous agents co-operatively performing concurrent machine runs. The behaviour of agents is determined by ASM programs, each consisting of a transition rule, which defines the set of possible runs. Each agent has its own partial view of a global state, which is defined by a set of static and dynamic functions and domains. Interaction among agents can be modelled by having non-empty intersections of partial views.
Please note that the term *agent* is used both in the RSDL description and in the ASM framework. Generally, an agent is an active entity having a behaviour and working in parallel with other agents. See also Section 2.1.2 for the ASM concept of agents and Section 3.7 for the RSDL agent concept.

1.5 Implementation of the Formal Semantics: Overview

The formal semantics clearly states the properties of RSDL. In order to find out whether the semantics is correct, however, it is necessary to check it against the language description and the intentions of the language designers. In order to work properly with the semantics, it is essential that its contents be easily accessible. This is best done using a correct implementation of the semantics[1].
The demand for an implementation was taken into account when designing the formal semantics. The following characteristics make an implementation possible.

- ASM is an operational technique: the basic state change primitive defined in ASM is an update, which is mostly the same as an assignment. All the other constructors merely define sets of assignments. It is important to note here that these sets must be finite to ensure executability, which is the case for the RSDL semantics.
- ASM is supported by tools: there are freely available implementations for ASM. We use the ASM work-bench from Paderborn [15]. Close contact with the developers of the ASM workbench ensures that the RSDL semantics can be processed by this tool.
- A special style of using ASM is applied for the definition of the RSDL semantics: Although ASM descriptions are executable in principle, there are some instructions that could cause problems. In particular sets can be constructed using all the features of first order predicate calculus. In the RSDL semantics, all set constructions are computable, i.e. they use only computable functions over finite domains.

To implement the semantics with minimum effort, existing tools are used wherever possible. The formal semantics is implemented according to the technology given in the next section.

[1] As the semantics implementation maps the semantics description into tools, all the statements below are also valid for the SDL semantics (implementation).

1.5.1 Implementation Technology Based on Abstract Syntax Trees

The tooling for the formal semantics is based on an abstract syntax tree (AST) representation of the input. As stated above, readily available tools are used to assist in implementing the semantics. Thus the compiler related parts are handled with the standard tools `lex` and `yacc`. After parsing, the further processing is defined over an abstract syntax tree representation of the input. This representation is generated by a tool called `kimwitu` [16].
The compiler technology of `kimwitu` is organised around an abstract representation of the input (see Figure 3 for an overview of the technology). In a top-level view, an input text is transformed to an output text using a transformation tool. This transformation tool consists of two main parts, namely a front-end tool and a back-end tool. The abstract syntax tree is the interface between those two parts.

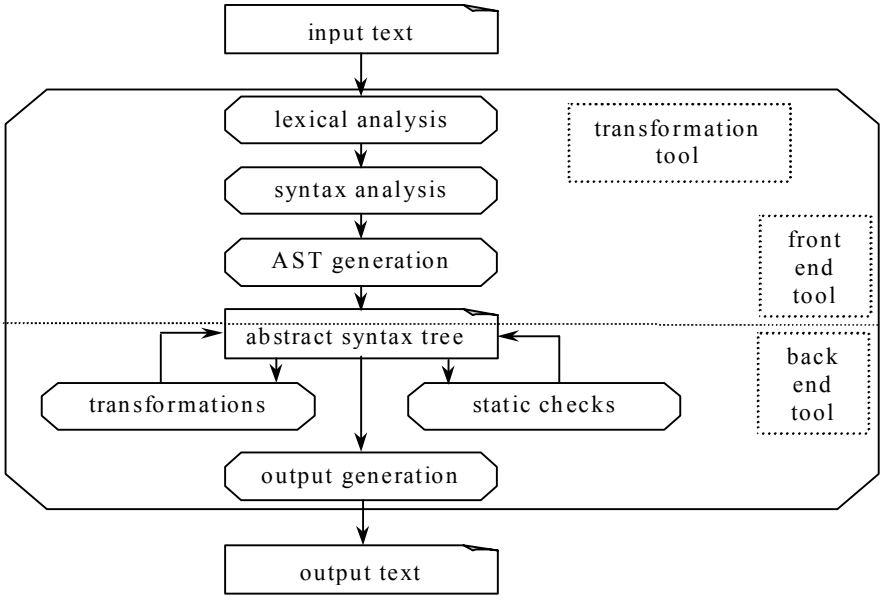


Figure 3: Overview of the Implementation Technology

The input text is first analysed by the front-end tool. This operation consists of a lexical analysis (done here with `lex`) and a syntax analysis (done with `yacc` in our case). Then an abstract syntax tree (AST) is generated using `kimwitu`. In fact, the `kimwitu`-generated constructor functions for building the AST are called from `yacc`.
Once the AST is constructed, the back-end tools can work on it. They check for the appearance of tree patterns and handle those. These tasks can be handled by the `kimwitu` *unparse rules*. Unparse rules permit the specification of tree patterns and what is to be done for them. Moreover, `kimwitu` allows the definition of functions over the tree in a simple way. This makes it possible to implement static checks: we simply define what the static check of one entity is with respect to its sub entities within the tree.
Another back-end task is to perform certain transformations on the tree in order to bring it into "normal form". Some constructs may be abbreviations of other constructs, and are transformed away. This is done by means of the `kimwitu` feature *rewrite rules*. The most simple form of rewrite rules is a mapping of one tree pattern into another tree pattern. However, sometimes it is necessary to describe more complex transformations, such as those that involve more than one location in the tree, or that compute some primitive elements out of other elements. For these cases `kimwitu` permits the definition of transformation rules that resort to arbitrary programming language functions.
Finally, once the input is analysed and transformed into normal form, one would like to process it or to generate code out of it. This can be achieved using `kimwitu` unparse rules again.

1.5.2 Implementing the Static Formal Part

The formal aspects of the lexis, the concrete syntax, and the abstract syntax are defined within the RSDL language description in Part 3. They are implemented using `lex`, `yacc` and `kimwitu`. This means that tools are constructed which transform the RSDL lexis into a `lex` file, the concrete syntax grammar into a `yacc` file, and the abstract syntax grammar into a `kimwitu` file.
These generated files are used to build an RSDL parser[2], i.e. a tool that takes an RSDL specification and generates an abstract syntax tree representation of it. This is the front-end tool as indicated in Figure 3.

[2] A parser is a program the analyses a text grammatically.

In the scope of RSDL, the back-end tool has to take care of the well-formedness conditions and the transformations, as can be seen in Figure 1. The static semantics conditions are defined as predicate calculus formulae over the AST. They are implemented as Boolean functions over the abstract syntax tree. The first step in the analysis of the AST is to call all the static condition formulae on all the locations in the tree where they are applicable. If any of these functions yields the value *False*, the specification is ill-formed.
In case of a correct specification, the next step is to transform the tree using the transformations as defined within the formal RSDL definition as rewrite rules. These rewrite rules are implemented as `kimwitu` rewrite rules.

1.5.3 Implementing the Dynamic Formal Semantics

The dynamic formal semantics consists of four different parts as explained in Section 1.4.2. The implementation of each part is discussed in turn.

Special Abstract Machine. The SAM consists of a number of declarations and definitions in ASM. These must be extracted from the formal definition and then transformed into a format suitable for a tool. We use the ASM workbench for the implementation.
The transformation of the ASM text into input for the ASM workbench is again done using the implementation methodology as explained in Section 1.5.1. This means that, first, an abstract syntax tree of the ASM definitions is generated, and then the input for the ASM workbench is generated using `kimwitu` unparse rules. Please note that this is done only once, since the SAM is independent of any particular RSDL specification.

Compilation. The compilation is defined as a function over the abstract syntax tree for RSDL behaviour. It generates sets of behaviour primitives in the sense of the SAM. It is straightforward to define an implementation for this function. This is a function over the abstract syntax tree, which can be expressed in `kimwitu`. The SAM behaviour representation that is generated from this function must be of an ASM format. This is achieved by unparsing the abstract tree.

Initialisation. The initialisation defines a set of initialisation programs and a pre-initial state. Like the SAM parts, the programs are transformed into the ASM workbench.
The pre-initial system state is coded as ASM predicates for the initial state. These predicates are not depending on the concrete specification. The only specification-dependent part of these conditions is a link to the actual abstract syntax tree that represents the abstracted specification. This part is generated from the `kimwitu` representation by means of an unparsing step.

Data Semantics. The semantics of the data part is defined in terms of the data interface. This interface is designed such that it only provides functional aspects of the data. All dynamic aspects are already covered by the compilation step. For RSDL, only basic predefined data are allowed, so the data part is fixed. It is handled in the same way as the SAM.

1.6 Roadmap of the Implementation

In order to better understand the structure of the book, please consider the roadmap of the implementation in Figure 4 below. The structure of the implementation is mapped to the chapters of the book. Please note that Section 2.2 Implementation Technology is used implicitly in many places, namely all those that have to do with the generation of files, and also within the generated compiler.
You may choose to read the book in any order—in this case, use the roadmap as an aid. If, for example, you want to study the lexical analysis, you should follow the path from the language description to the lexer. If you want to find out about the ASM aspects, you should consider the arrows leading to the ASM workbench.
If this book is used as a textbook for software engineering, Part 2 and Part 5 are essential.
If this book is used as a textbook for formal semantics definition, Part 2, Part 3 and Part 4 are essential.
SDL experts may skip the RSDL language description in Part 3 and only use the short description in Section 3.1.
Please note that `kc` is used as an abbreviation for `kimwitu` in Figure 4.

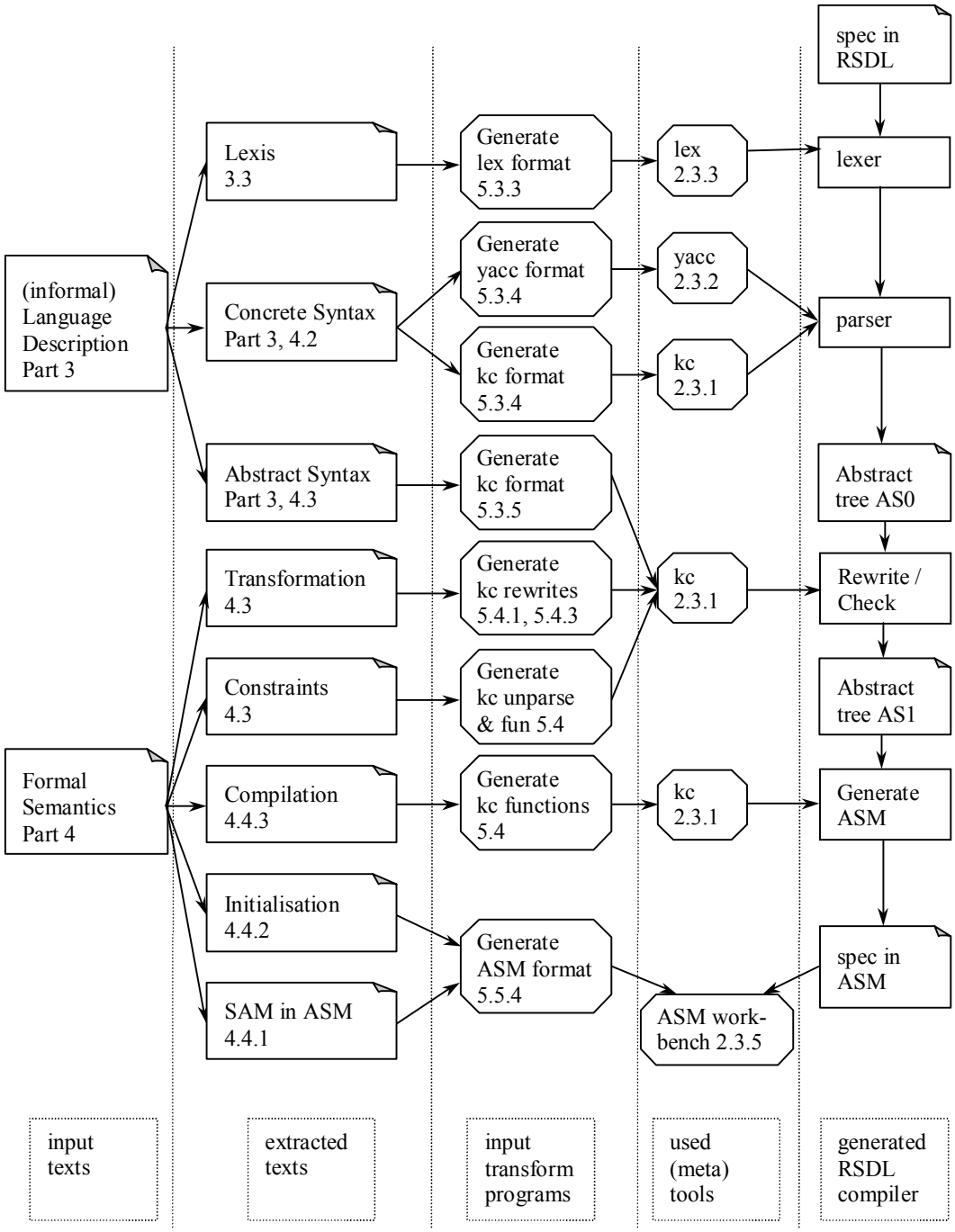


Figure 4: Roadmap of the Implementation

Part 2: BASICS

This part is devoted to mostly informal descriptions of the elements that are used later. Section 2.1 introduces a variant of Abstract State Machines as it is used to formalise the RSDL semantics in Part 4. Section 2.2 surveys the implementation technology that is used within Part 5 to implement the formal semantics. Section 2.3 describes in more depth the tools that are used for the implementation.

2.1 ASM

In this section, the basics of *Abstract State Machines* (*ASM*) and the notation to define ASM models that is used in this document are explained. The objective here is to provide an intuitive understanding of the formalism; for a rigorous definition of the mathematical foundations of ASM, the reader is referred to [1] and [2]. Further references on ASM can be found on the ASM Web Pages [4].
The ASM model used to define the dynamic semantics of RSDL is explained in several steps. Firstly, the *single-agent ASM model* is treated (Section 2.1.1). Next, this model is extended to cover *multi-agent systems* (Section 2.1.2). Then, *open systems*, i.e. systems interacting with an environment they do not control, are addressed by adding the notion of *external world* (Section 2.1.3). Finally, the model is extended by introducing a notion of *real-time behaviour* (Section 2.1.4). To illustrate these steps, an ASM model for a simple system is developed, step by step. The final ASM model of this system is summarised in Section 2.1.5. Additional notation used to define the dynamic semantics of RSDL is explained in Section 2.1.6.

EXAMPLE (Informal Description):
In order to illustrate the ASM model, a simple resource management system *RMS* consisting of *a group of agents* competing for a *resource*, for instance, some device or service, is defined. Informally, this system is characterised as follows:

- There is a set of *tokens* used to grant *exclusive* or *non-exclusive* (*shared*) access to the resource.
- Depending on whether the desired access mode is exclusive or shared, an agent must own all tokens or one token, respectively, before it may access the resource.
- An agent is *idle* when not competing for a resource, *waiting* when trying to obtain access to the resource, or *busy* when owning the right to access the resource.
- A busy agent releases the resource when it is no longer needed, as indicated by a *stop condition*. On releasing the resource, all tokens owned by the agent are returned.
- Stop conditions are only indicated when an agent is busy.
- Initially, all agents are idle, and all tokens are available.

The system will be defined step by step, as the explanations of the ASM model proceed, starting with a single-agent ASM. The final ASM model of this system is summarised in Section 2.1.5.

2.1.1 Single-Agent ASM

A *single-agent Abstract State Machine M* is defined over a given *vocabulary V* by its *states S*, its *initial states* S_0, and its *program P*. These items will be explained in the following subsections.

2.1.1.1 Vocabulary

The vocabulary (or signature) *V* denotes a finite set of *function names*, *predicate names*, and *domain names*, each of a fixed arity. Names in *V* are classified as *basic* or *derived*, and further distinguished into *static* or *dynamic* (see Figure 5). The meaning associated with these classifications will be explained in subsequent subsections.

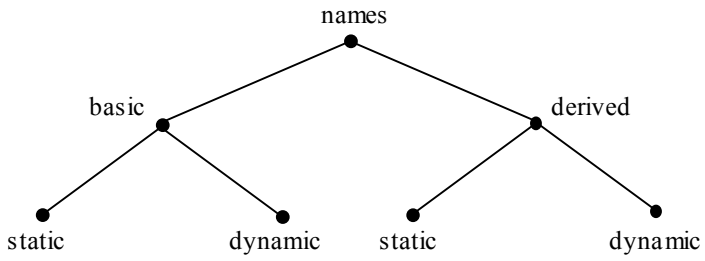


Figure 5: Classification of ASM Names

V is declared when defining an ASM, except for a subset of *predefined names*. This subset includes, for instance, the equality sign, the nullary predicate names *True*, *False*, the nullary function name *undefined*, the domain names *BOOLEAN*, *INT* and *REAL*, as well as the names of frequently used standard functions (such as Boolean operations $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$, and set operations $\subseteq$, $\cup$, $\cap$, $\in$, $\notin$, etc.). Predefined names are listed in Section 2.1.6.
To *declare* names when defining a concrete ASM, we use the following notational conventions:

- *Domain names* are written in capitalised italics (as in *AGENT*), except when denoting a non-terminal of the abstract grammar. Abstract grammar domain names are written like non-terminals, in italics, hyphenated, and starting with a capital (e.g. *Agent-identifier*). A domain name *D* is declared by **domain** *D*.
- *Function names* are written in italics, usually starting with a small letter (as in *mode*). A function name *f* is declared by f: $D_1 \times D_2 \times ... \times D_n \rightarrow D_0$, where *n* is the arity of *f*, and $D_0, D_1, D_2, ..., D_n$ are domains.
- *Predicate names* are written in italics, starting with a capital letter (as in *Available*). A predicate name *P* is a function name with the result domain *BOOLEAN*, i.e. it is declared by P: $D_1 \times D_2 \times ... \times D_n \rightarrow$ *BOOLEAN*.
- *Basic static names* are preceded by the keyword **static**, when they are declared.
- *Dynamic names* are preceded by one of the keywords **controlled, monitored**, or **shared** when they are declared.
- Names declared without a preceding keyword are *derived names* by default.

EXAMPLE (Vocabulary):
To define an ASM model of the system *RMS*, assume a vocabulary *V* including the following names:

static domain *AGENT*
static domain *TOKEN*
domain *MODE*

shared *mode*: *AGENT* $\rightarrow$ *MODE*
controlled *owner*: *TOKEN* $\rightarrow$ *AGENT*
static *ag*: $\rightarrow$ *AGENT*

Idle: *AGENT* $\rightarrow$ *BOOLEAN*
Waiting: *AGENT* $\rightarrow$ *BOOLEAN*
Busy: *AGENT* $\rightarrow$ *BOOLEAN*
Available: *TOKEN* $\rightarrow$ *BOOLEAN*

monitored *Stop*: *AGENT* $\rightarrow$ *BOOLEAN*

The domain names *AGENT*, *TOKEN*, and *MODE* are introduced to represent the (single) agent of the system, the set of tokens, and the different access modes (exclusive, shared), respectively. The function names *mode* and *owner* are dynamic, they are used to denote the current access mode of an agent and the current owner of a token, respectively. The nullary function name *ag* refers to a value of the domain *AGENT*. *Idle*, *Waiting*, *Busy*, and *Available* are derived, dynamic predicate names. *Stop* denotes a monitored predicate name, which will be explained later.

2.1.1.2 States

A state $s \in S$ is given by assigning a meaning, called *interpretation*, to the names in *V* over an infinite set, called *base set of M* (to which we refer by the predefined domain name *X*). This is achieved by associating basic domain names, function names and predicate names with domains, functions, and predicates, respectively. The interpretation of derived names follows from the interpretation of basic names. Note that the base set is the same for all states of *M*. It is required that *True*, *False* and *undefined* denote distinct elements of the base set. Predefined operations have their usual interpretation.
Recall that names are classified as static or dynamic. If classified as static, names are required to have the same interpretation in all states of *M*. Otherwise, they may have different interpretations in different states of *M*. Thus, the states *S* of *M* are given by the set of all interpretations of the names in *V* over the base set of *M* that comply with these and other explicitly stated constraints.
Strictly speaking, all functions are *total* functions on the base set of *M*. To imitate *partial functions*, "undefined" function values are marked by the distinguished element *undefined*. Predicates only yield one of the values *True* or *False*, i.e., they must not be partial.

Every state has an infinite number of *reserve elements* that can be used to extend domains dynamically (see Section 2.1.1.6). By definition, the reserve elements of a state are all those elements of the base set which are neither identified by a function nor contained in one of the domains, i.e. they are not reachable using the vocabulary.

2.1.1.3 Derived Names

Let *DerivedName* be an n-ary name, and let $Formula(v_1,...,v_n)$ denote a formula of the domain D_0 with free variables $v_1,...,v_n$ of domains $D_1,...,D_n$, $n \geq 0$. The general form of a *derived name definition* is:

$DerivedNameDefinition ::= DerivedName(v_1{:}D_1,...,v_n{:}D_n){:}D_0 =_{def} Formula(v_1,...,v_n)$

The result domain D_0 is omitted in case of a derived domain definition.

The meaning of derived names follows from the meaning of basic names, and is defined in terms of *formulae* (see Section 2.1.6).

EXAMPLE (Definitions):
The following derived names are defined in the RMS:

$MODE =_{def} \{exclusive, shared\}$

$Idle(a{:}AGENT){:}\ BOOLEAN \quad =_{def} \quad a.mode = undefined \wedge \forall t \in TOKEN{:}\ t.owner \neq a$
$Waiting(a{:}AGENT){:}\ BOOLEAN \quad =_{def} \quad a.mode \neq undefined \wedge \forall t \in TOKEN{:}\ t.owner \neq a$
$Busy(a{:}AGENT){:}\ BOOLEAN \quad =_{def} \quad a.mode \neq undefined \wedge \exists t \in TOKEN{:}\ t.owner = a$
$Available(t{:}TOKEN){:}\ BOOLEAN \quad =_{def} \quad t.owner = undefined$

An agent *a* is, for instance, idle iff the function *mode* yields the value *undefined* for that agent, and *a* does not hold any token. A token *t* is available iff no agent is holding *t*.

For an improved readability, we use a *"."-notation* for unary functions and predicates. For instance, we write *a.mode*, which is equivalent to writing *mode*(*a*).

2.1.1.4 Initial States

The set of *initial states* $S_0 \subseteq S$ can be defined by constraints imposed on domains, functions, and predicates. These constraints are required to hold in the *first* state of each *run of M* (see Section 2.1.1.5). Initial constraints have the following general form:

initially *ClosedFormula*

EXAMPLE (Initial States):
The following constraints define the set of initial states of the system *RMS*:

initially $AGENT = \{ag\}$
initially $(\forall a \in AGENT{:}\ a.Idle) \wedge (\forall t \in TOKEN{:}\ t.Available)$

The first constraint defines *AGENT* to consist of a single element *ag*. Because *AGENT* is static, this formula does always hold. The last constraint expresses that initially, the agent of *RMS* is idle (*a.mode = undefined*), and all tokens are available (*t.owner = undefined*). Note that no constraint on *Stop* is defined.

2.1.1.5 State Transitions and Runs

Recall that a (global) state $s \in S$ is given by an interpretation of the names in *V* over the base set of *M*. State transitions can be defined in terms of partial reinterpretations of dynamic domains, functions, and predicates. This gives rise to the notions of *location* as a conceptual means to refer to parts of global states, and of *update* to describe state changes.
A *location of a state s of M* is a pair $loc_s = \langle f, seq \rangle$, where *f* is a dynamic name in *V*, and *seq* is a sequence of elements of the base set according to the arity of *f*. An *update of s* is a pair $\delta_s = \langle loc_s, new \rangle$, where *new* identifies an element of the base set as the new value to be associated with the location loc_s. To *fire* δ_s means to transform *s*

into a state s' of M such that $f_s(seq) = new$, while all other locations loc'_s of s, $loc'_s \neq loc_s$, remain unaffected. In other words, firing an update modifies the interpretation of a state in a well-defined way.
The potential behaviour of a single-agent ASM is captured by a *program P*, which is defined by a *transition rule* (see Section 2.1.1.6 and Section 2.1.1.8). For each state $s \in S$, a program P of M defines an *update set* $\Delta_s(P)$ as a finite set of updates of s. $\Delta_s(P)$ is *consistent*, iff it does not contain any two updates δ_s, δ'_s such that $\delta_s = <loc_s, new>$, $\delta'_s = <loc_s, new'>$, and $new \neq new'$. The *firing of a consistent update set* $\Delta_s(P)$ in state s means to fire all its members simultaneously, i.e. to produce in one atomic step a new state s' such that for all locations $loc_s = <f, seq>$ of s, $f_s(seq) = new$, if $<<f, seq>, new> \in \Delta_s(P)$, and $f_s(seq) = f_s(seq)$ otherwise, and is called *state transition*. Firing an inconsistent update set has no effect, i.e., $s' = s$.[3]
The behaviour of a single-agent ASM M is modelled through (finite or infinite) *runs of M*, where a *run* is an interleaved sequence of states and moves of the form

$$s_0 \xrightarrow{\Delta_{s0}(P)} s_1 \xrightarrow{\Delta_{s1}(P)} s_2 \xrightarrow{\Delta_{s2}(P)} \ldots \qquad \begin{matrix}\text{Moves}\\ \text{States}\end{matrix}$$

such that $s_0 \in S_0$, and s_{i+1} is obtained from s_i, for $i \geq 0$, by firing $\Delta_{si}(P)$ on s_i, where $\Delta_{si}(P)$ denotes an update set defined by the program P of M on s_i (see Section 2.1.1.8). The meaning of an ASM is defined to be the set of all its runs.

2.1.1.6 Transition Rules

Transition rules specify update sets over ASM states. Rules are formed from elementary rules using various rule constructors. There is one elementary transition rule, called *update instruction*, and a set of rule constructors:

update instruction

Rule ::= $f(t_1,...,t_n) := t_0$ $\quad (n \geq 0)$

Here, f is a basic and dynamic name of V, and $t_0,t_1,...,t_n$ are terms over V identifying, for a given state s, the location $loc = <f,<s(t_1),..., s(t_n)>>$ to be changed and the new value $s(t_0)$ to be assigned. Hence, the above update instruction specifies the update set $\{<<f,<s(t_1),..., s(t_n)>>, s(t_0)>\}$, consisting of a single update. Note that only locations identified by basic and dynamic names may be modified by an update instruction.

EXAMPLE (Update Instruction):
Let t be a variable denoting a token, and ag be an agent.

$t.owner := ag$	specifies the update set $\{<<owner, <s(t)>>, s(ag)>\}$
$ag.mode := undefined$	specifies the update set $\{<<mode, <s(ag)>>, s(undefined)>\}$

The construction of complex transition rules out of elementary update instructions is recursively defined by means of *ASM rule constructors*. For the ASM model applied to define the RSDL semantics, six constructors are used. These constructors are listed below, with an informal description of their meaning. Here, *Rule*, $Rule_1$ denote transition rules, g denotes a Boolean term, and $v,v_1,...,v_n$ denote variables over the base set of M. The scope of a rule constructor is expressed by appropriate keywords, and can additionally be indicated by indentation. The closing keywords can be omitted, if no confusion arises. If closing keywords are omitted, the corresponding constructor extends as much as possible, but not over the next **where**.

if-then-constructor

Rule ::= **if** g **then**
  $Rule_1$
[**else**
  $Rule_2$]
[**endif**]

The update set defined by *Rule* in state s is defined to be the update set of $Rule_1$ or $Rule_2$, depending on the value of g in state s. Without the optional **else**-part, the update set defined by *Rule* is the update set of $Rule_1$ or the empty update set. Sometimes **elseif** is used as abbreviation for **else if**.

[3] Actually, an inconsistent update set indicates an error in the semantic model. The ASM semantics ensures that such errors do not destroy the notion of state.

do-in-parallel-constructor

Rule ::= [**do in-parallel**]
Rule$_1$
...
Rule$_n$
[**enddo**]

The update set defined by *Rule* in state *s* is defined to be the union of the update sets of *Rule*$_1$ through *Rule*$_n$. This implies that the order in which transition rules belonging to the same block are stated is irrelevant. For brevity, the keywords **do in-parallel** and **enddo** may be omitted, where no confusion arises. Hence, an ASM program often appears as a collection of rules rather than a monolithic block rule.

do-forall-constructor

Rule ::= **do forall** v: $g(v)$
Rule$_0(v)$
[**enddo**]

The effect of *Rule* is that *Rule*$_0$ is fired simultaneously for all elements v of the base set of M for which the Boolean condition $g(v)$ holds in state s, where v is a free variable in *Rule*$_0$. More precisely, Δ_s(*Rule*) is the union of all update sets Δ_s(*Rule*$_0(v)$) such that $g(v)$ holds in state s. Note that update sets are required to be finite, therefore, $g(v)$ must hold for a finite number of values only.

choose-constructor

Rule ::= **choose** v: $g(v)$
Rule$_0(v)$
[**endchoose**]

The effect of *Rule* is that *Rule* is fired for some element v of the base set of M for which the Boolean condition $g(v)$ holds in state s, where v is a free variable in *Rule*$_0$. More precisely, Δ_s(*Rule*) is some update set Δ_s(*Rule*$_0(v)$) such that $g(v)$ holds in state s, or the empty update set if no such v exists.

extend-constructor[4]

Rule ::= **extend** D **with** $v_1,...,v_n$
Rule$_0(v_1,...,v_n)$
[**endextend**]

The effect of *Rule* when fired in some state s is that n reserve elements of s (see Section 2.1.1.2) are imported into the dynamic domain D (while being removed from the reserve), that $v_1,...,v_n$ become bound to one of the imported elements each, and that *Rule*$_0(v_1,...,v_n)$ is fired.
The *extend* constructor can be used to mimic object-based ASM definitions, where objects are dynamically created. Thus, for each object to be created, an element from the reserve is assigned to the corresponding domain, and initialised.

let-constructor

Rule ::= **let** v = *expression* **in**
Rule$_0(v)$
[**endlet**]

The effect of *Rule* when fired in some state s is that v is bound to the value of *expression*, and that *Rule*$_0$ is fired with this value.

[4] Strictly speaking, *extend* can be defined in terms of the *import* constructor (not shown here); however, since the *import* constructor will not be used otherwise, we simplify the introduction of the ASM notation at this point.

EXAMPLE (Transition Rule):
The following transition rule defines the behaviour of agent *ag* when requesting shared access, i.e. when *ag.mode = shared*. To state the rule, the if-then-constructor, the choose-constructor, and the update instruction are applied.

if *ag.mode = shared* ∧ *ag.Waiting* **then**
  choose *t*: *t* ∈ *TOKEN* ∧ *t.Available*
    t.owner := *ag*

The precise meaning of the rule is determined by its update set with respect to a state *s*, which is either {<<*owner*, <*s*(*t*)>>, *s*(*ag*)>} for some token *s*(*t*) available in *s*, if all further predicates stated in the if-then-constructor hold in *s*, or empty otherwise.

2.1.1.7 Abbreviations

Rules can be structured using *abbreviations*, consisting of *rule macros* and *derived names*, that may have parameters. This allows for hierarchical definitions, and the stepwise refinement of complex rules, which supports the understanding of ASM model definitions. The following syntax constructs are introduced for defining and using abbreviations.

abbreviation-definition

Abbreviation ::= *RuleMacroDefinition*
  | *DerivedNameDefinition*

Derived names are introduced as explained in Sections 2.1.1.3, i.e. by definition.

rule-macro-definition

Let $Rule_0$ denote a transition rule with free variables $v_1,...,v_n$, $n \geq 0$. The general form of a rule macro definition is:

RuleMacroDefinition ::= $RuleMacroName(v_1,...,v_n) \equiv Rule_0(v_1,...,v_n)$

Rule macro names are, by convention, written in small capitals, with a leading capital letter (as in SHAREDACCESS).

where-part

By default, abbreviations have a global scope. However, the scope of an abbreviation can also be restricted to a particular transition rule $Rule_0$ by using the where-part.

Rule ::= $Rule_0$
  where
    $Abbreviation^+$
  endwhere

Similarly, an abbreviation can be declared local to an expression.

rule-macro-application

Rule macros are applied in transition rules as follows:

Rule ::= $RuleMacroName(t_1,...,t_n)$

Formally, rule macros are syntactical abbreviations, i.e., each occurrence of a macro in a rule is to be replaced textually by the related macro definition (replacing formal parameters by actual parameters).

EXAMPLE (Rule Macro):

The transition rule from the previous example can be stated using rule macros, and be defined as a macro itself. Here, SHAREDACCESS is a macro definition with global scope that can be used in other places of the ASM model definition. GETTOKEN is a parameterised macro definition with a local scope restricted to the rule SHAREDACCESS, with formal parameter *a*. When GETTOKEN is applied in SHAREDACCESS, *a* is replaced by the actual parameter *ag*.

```
SHAREDACCESS ≡
    if ag.mode = shared ∧ ag.Waiting then
        GETTOKEN(ag)
    where
        GETTOKEN(a) ≡
            choose t: t ∈ TOKEN ∧ t.Available
                t.owner := a
    endwhere
```

2.1.1.8 Single-Agent ASM Programs

A *single-agent ASM program P* is given by a framed *transition rule* (or *rule* for short) of the following form:

Rule

As already mentioned, rule macro definitions may either have local or global scope. To have a global scope, the macro definitions can be given outside the ASM program, and can be applied in the ASM program.
At the moment, it is sufficient to define one ASM program, which is statically associated with the agent of the single-agent ASM model. In the next section, we will allow to define several ASM programs, and associate them with different agents dynamically.

EXAMPLE (ASM Program):

The ASM program *P* of the system *RMS* is defined as follows:

```
do in-parallel
    SHAREDACCESS
    EXCLUSIVEACCESS
    RELEASEACCESS
enddo
where
    SHAREDACCESS ≡
        if ag.mode = shared ∧ ag.Waiting then
            choose t: t ∈ TOKEN ∧ t.Available
                t.owner := ag
    EXCLUSIVEACCESS ≡
        if ag.mode = exclusive ∧ ∀t ∈ TOKEN: t.Available then
            do forall t: t ∈ TOKEN
                t.owner := ag
    RELEASEACCESS ≡
        if ag.Busy ∧ ag.Stop then
            do in-parallel
                ag.mode := undefined
                do forall t: t ∈ TOKEN ∧ t.owner = ag
                    t.owner := undefined
endwhere
```

The ASM program is defined by a single transition rule as shown in the frame. The transition rule uses the do-in-parallel-constructor and 3 rule macros, which results in a hierarchical rule definition.

2.1.2 Multi-Agent ASM

Mathematical modelling of concurrent and reactive systems requires to extend the single-agent ASM model. In this section, the concept of *multi-agent ASM*, which generalises the single-agent ASM model presented in Section 2.1.1, is explained.

A *multi-agent Abstract State Machine M* is defined over a given *vocabulary V* by its *states S*, its *initial states* S_0, its agents *A*, and its *programs P*. These items will be explained in the following subsections, as far as they differ from the single-agent ASM model.

2.1.2.1 Vocabulary

The vocabulary *V* of a multi-agent ASM *M* includes the predefined domains *AGENT* and *PROGRAM*, representing *M*'s set *A* of agents and *P* of ASM programs, respectively. *AGENT* is dynamic in general, while *PROGRAM* is required to be static.

Furthermore, *V* includes a function name *program*: *AGENT* → *PROGRAM*, which is dynamic in general, and a special nullary function *Self* (see Section 2.1.2.2). An agent *a* is inactive when *a.program = undefined*.

2.1.2.2 Agents and Runs

A multi-agent ASM may have any finite number of agents, where this number may be dynamically varying. The behaviour of each agent is determined by some program from a static collection of ASM programs, where each program is defined by a single transition rule. Agents operate concurrently by running their ASM programs, and interact asynchronously through globally shared locations of the state, where two or more agents may read and write the same location. The runs of the multi-agent ASM are then defined in terms of *partially ordered runs* (see [1]), which addresses true concurrency directly instead of just approximating concurrency by an interleaving model.

To assign a behaviour to an agent of *M*, the distinguished function *program* (see Section 2.1.2.1) yields, for each agent *a* of *M*, the program of *P* to be executed by *a*. The function *program* may allow to define (or to redefine) the behaviour of agents dynamically; it is thereby possible to create new agents at run time. Regarding a given state *s* of *M*, the agents associated with *s* are those elements *a* from the underlying base set such that *a.program* identifies some program of *P*.

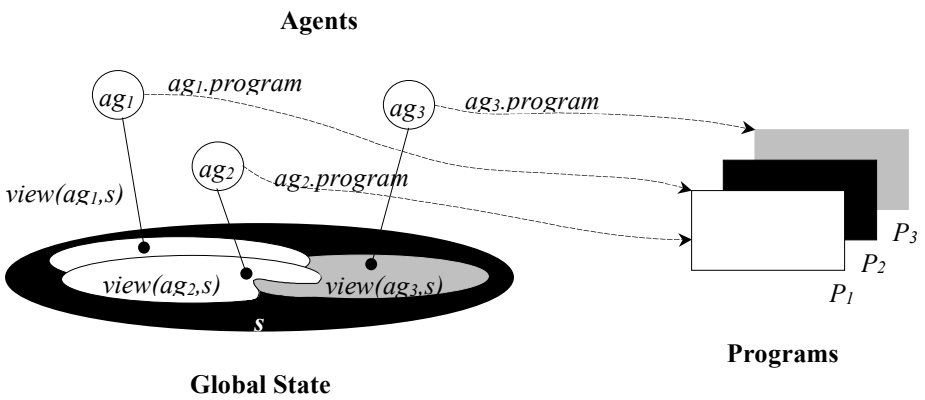
A special nullary function *Self* serves as a *self reference* identifying the agent calling *Self*:

monitored *Self*: → *AGENT*

For every agent, *Self* has a different interpretation. By using *Self* as an additional function argument, each agent *a* can have its own partial view of a given global state of *M* on which it fires the rule in *a.program*.

EXAMPLE (OVERVIEW OF MULTI-AGENT ASM):

In the following figure, a particular multi-agent ASM *M*, consisting of three agents ag_1, ag_2, and ag_3 is illustrated. The function *program* associates, with each agent, one of the ASM programs P_1, P_2, and P_3. Here, ag_1 and ag_2 are assigned the same program. Program P_2 is currently not associated with any agent, however, this may change during execution, as *program* is a dynamic function. Each agent has its own *partial view* on a given global state *s* of *M*, in which it fires the rule of its current program. In the figure, this view is illustrated by the function *view*, which yields, for each agent, its local and its shared state. In fact, the current view of each agent is determined implicitly by the ASM model definition, including the ASM programs.



The semantic model of concurrency models the behaviour of multi-agent ASMs in terms of partially ordered runs. A *partially ordered run* represents a certain class of (admissible) runs of a multi-agent ASM by restricting non-determinism with respect to the order of so-called *moves*, i.e., a step of a single agent. The moves of an agent are always linearly ordered, whereas moves of different agents need only be ordered if they are causally dependent. Partially ordered runs directly support the modelling of *true* concurrency.

Partially Ordered Runs
Regarding the moves of an individual agent, these are linearly ordered, whereas moves of different agents need only be ordered in case that they are not *independent* of each other. Intuitively, independent moves model concurrent actions which are incomparable with regard to their order of execution. The precise meaning of independence is implied by the coherence condition in the formal definition of partially ordered runs (adopted from [1]).
A run ρ of a multi-agent ASM M is given by a triple $<\Lambda, A, \sigma>$ satisfying the following four conditions:

1. Λ is a partially ordered set of moves, where each move has only a finite number of predecessors;
2. A is a function on Λ associating agents to moves such that the moves of any single agent of M are linearly ordered;
3. σ assigns a state of M to each initial segment Y of Λ, where $\sigma(Y)$ is the result of performing all moves in Y;
4. if y is a maximal element in a finite initial segment Y of Λ and $Z = Y - \{ y \}$, then $A(y)$ is an agent in $\sigma(Z)$ and $\sigma(Y)$ is obtained from $\sigma(Z)$ by firing $A(y)$ at $\sigma(Z)$ (*coherence condition*).

Implications of Partially Ordered Runs
Partially ordered runs have certain characteristic properties which can be stated in terms of *linearisations*[5] of partially ordered sets. A linearisation of a partially ordered set Λ is a linearly ordered set Λ' with the same elements such that if $y < z$ in Λ then $y < z$ in Λ'. Accordingly, the semantic model of concurrency as implied by the notion of partially ordered run can further be characterised as follows [1]:

- All linearisations of the same finite initial segment of a run of M have the same final state.
- A property holds in every reachable state of a run ρ of M if and only if it holds in every reachable state of every linearisation of ρ.

2.1.2.3 Multi-Agent ASM Programs

A *multi-agent ASM program* $p \in P$ is given by a *program name* and a *transition rule* (or *rule* for short). The program name uniquely identifies p, and is represented by a unary static function[6]. Programs are stated in the following form:

ASM-PROGRAM:

```
Rule
```

Program names are, by convention, hyphenated and written in small capitals, with a leading capital letter (as in RESOURCE-MANAGEMENT-PROGRAM).

EXAMPLE (ASM Program):

The multi-agent ASM program of the system *RMS* is defined as follows:

RESOURCE-MANAGEMENT-PROGRAM:

```
do in-parallel
    SHAREDACCESS
    EXCLUSIVEACCESS
    RELEASEACCESS
enddo
where
    SHAREDACCESS ≡
        if Self.mode = shared ∧ Self.Waiting then
            choose t: t ∈ TOKEN ∧ t.Available
                t.owner := Self
```

[5] Sometimes, a linearisation is called *interleaving*.

[6] Strictly speaking, the program names of M are represented by a distinguished set of elements from the base set.

```
EXCLUSIVEACCESS ≡
    if Self.mode = exclusive ∧ ∀t ∈ TOKEN: t.Available then
        do forall t: t ∈ TOKEN
            t.owner := Self
RELEASEACCESS ≡
    if Self.Busy ∧ Self.Stop then
        do in-parallel
            Self.mode := undefined
            do forall t: t ∈ TOKEN ∧ t.owner = Self
                t.owner := undefined
endwhere
```

The multi-agent ASM program has the name RESOURCE-MANAGEMENT-PROGRAM, and is defined as the single-agent ASM program before, with one difference: all occurrences of *ag* have been replaced by calls of the function *Self*. This allows to associate the program with different agents, while accessing the local state of these agents.

The domain *PROGRAM* is implicitly defined as follows.

$$PROGRAM =_{def} \{PROGRAM_1,...,PROGRAM_n\}$$

Here $PROGRAM_1,...,PROGRAM_n$ are the names of the programs that are defined in the ASM model.

2.1.3 The External World

Following an *open system view*, interactions between a system and the external world, e.g. the environment into which the system is embedded, are modelled in terms of various interface mechanisms. Regarding the reactive nature of distributed systems, it is important to clearly identify and precisely state

- preconditions on the expected behaviour of the external world, and
- how external conditions and events affect the behaviour of an ASM model.

This is achieved through a classification of *dynamic* ASM names into three basic categories of names[7], which extends the classification of names shown in Figure 6.

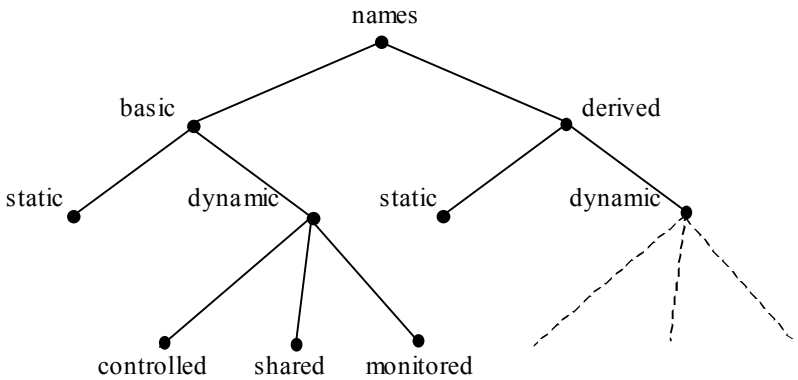


Figure 6: Extended classification of ASM names

- *controlled functions*
  These functions can only be modified by agents of the ASM system, according to the executed ASM programs. Controlled functions are preceded by the keyword **controlled** at their point of declaration.
- *monitored functions*
  These functions can only be modified by the environment, but can be evaluated by ASM agents. Thus, a monitored function may change its interpretation from state to state in an arbitrary way, unless this is restricted by *integrity constraints* (see below). Monitored functions are preceded by the keyword **monitored** at their point of declaration.

[7] Strictly speaking, ASM states are mathematical objects which do not contain relations (domains and predicates), rather they deal with relations through characteristic functions; therefore, it suffices here to concentrate on functions.

- *shared functions*
  These functions may be altered by the environment as well as by the ASM agents. Therefore, an *integrity constraint* on shared functions is that no interference with respect to mutually updated locations must occur. Hence, it is required that the environment itself acts like an ASM agent (or a collection of ASM agents). Shared functions are preceded by the keyword **shared** at their point of declaration.

EXAMPLE (External World):
The vocabulary *V* of the system *RMS* is extended by a classification of dynamic functions and predicates:

shared *mode*: $AGENT \rightarrow MODE$
controlled *owner*: $TOKEN \rightarrow AGENT$

monitored *Stop*: $AGENT \rightarrow BOOLEAN$

The function *mode*, which determines the current access mode, is shared. It may be affected by external set operations, switching it to one of the values *exclusive* or *shared*. Furthermore, it is reset internally when the resource is released (see Section 2.1.2.3).
The predicate *Stop* represents an external stop request, such as an interrupt, and therefore is monitored.

In general, the influence of the environment on the system through shared and monitored names may be completely unpredictable. However, preconditions on the environment behaviour may be expressed by stating *integrity constraints*, which are required to hold in *all* states and runs of *M*. Note that we use integrity constraints only to express preconditions on the environment behaviour, but *not* properties the system is supposed to have. Integrity constraints are of the following form:

IntegrityConstraint ::= **constraint** *ClosedFormula*

EXAMPLE (Integrity Constraints):
The following integrity constraint states that stop requests are only generated for busy agents:

constraint $\forall a \in AGENT: (a.Stop \Rightarrow a.Busy)$

2.1.4 Real-time Behaviour

By introducing a notion of *real time* and imposing additional constraints on runs, we obtain a specialised class of ASMs, called *multi-agent real-time ASM*, with agents performing *instantaneous* actions in *continuous* time. Essentially, this means that agents fire their rules at the moment they are enabled.
To incorporate real-time behaviour into the underlying ASM execution model, we introduce a nullary monitored function *currentTime* returning real values. Intuitively, *currentTime* refers to the physical time. As an integrity constraint on the nature of physical time, it is assumed that *currentTime* changes its values monotonically increasing over ASM runs.

monitored *currentTime*: $\rightarrow REAL$

Consider a given vocabulary V containing *REAL* (but not *currentTime*) and let V^+ be the extension of V with the function symbol *currentTime*. Restrict attention to V^+-states where *currentTime* evaluates to a nonnegative real number. One can then define a run R of the resulting machine model as a mapping from the interval $[0,\infty)$ to states of vocabulary V^+ satisfying the following *discreteness requirement*, where $\sigma(t)$ denotes the reduct[8]) of $R(t)$ to V:

1. for every $t \geq 0$, *currentTime* evaluates to t at state $R(t)$;
2. for every $\tau > 0$, there is a finite sequence $0 = t_0 < t_1 < \ldots < t_n = \tau$ such that
   if $t_i < \alpha < \beta < t_{i+1}$ then $\sigma(\alpha) = \sigma(\beta)$.

Exploiting the discreteness property, one effectively obtains some finite representation (*history*) for every finite (sub-) run by abstracting from those states which are not considered as significant such that they contribute any relevant information to a behaviour description. In particular, state changes that only mean increasing *currentTime* are not considered. From the above definition of run it follows that only finitely many states are left.

[8] That is, for a given value t, we obtain $\sigma(t)$ from $R(t)$ by ignoring the interpretation of the function name *currentTime*.

Global System Time
In RSDL, the *global system time* is represented by the expression **now** assuming that values of **now** increase monotonically over system runs. In particular RSDL allows to have the same value of **now** in two or more consecutive system states. Building on the concept of multi-agent real-time ASM, we model this behaviour using a unary monitored function *clock* in combination with *currentTime*. The role of *clock* is to define a mapping from externally controlled values of physical time to internally observable values of global system time. Accordingly, *clock* determines time values taking the output of *currentTime* as its input.

monitored *clock*: *REAL* → *REAL*

There are two integrity constraints on the behaviour of *clock*:
1. *clock* values change monotonically increasing;
2. *clock* values are not bounded, i.e. the time does never stop.

We introduce the derived function *now* as an abbreviation to refer to the observable global system time.

now: *REAL* $=_{\text{def}}$ *clock*(*currentTime*)

2.1.5 Example: The System *RMS*

In this section, we assemble the pieces of the ASM model definition of the system *RMS* into their final version. For better reference, we also repeat the informal description.

Informal Description
In order to illustrate the ASM model, a simple resource management system *RMS* consisting of *a group of agents* competing for a *resource*, for instance, some device or service, is defined. Informally, this system is characterised as follows:

There is a set of *tokens* used to grant *exclusive* or *non-exclusive* (*shared*) access to the resource.

Depending on whether the desired access mode is exclusive or shared, an agent must own all tokens or one token, respectively, before he may access the resource.

An agent is *idle* when not competing for a resource, *waiting* when trying to obtain access to the resource, or *busy* when owning the right to access the resource.

A busy agent releases the resource when it is no longer needed, as indicated by a *stop condition*. On releasing the resource, all tokens owned by the agent are returned.

Initially, all agents are idle, and all tokens are available.

Vocabulary

static domain *TOKEN*
MODE $=_{\text{def}}$ {*exclusive*, *shared*}

shared *mode*: *AGENT* → *MODE*
controlled *owner*: *TOKEN* → *AGENT*

monitored *Stop*: *AGENT* → *BOOLEAN*

Derived Names

Idle(*a*:*AGENT*): *BOOLEAN* $=_{\text{def}}$ $a.mode = undefined \wedge \forall t \in TOKEN: t.owner \neq a$
Waiting(*a*:*AGENT*): *BOOLEAN* $=_{\text{def}}$ $a.mode \neq undefined \wedge \forall t \in TOKEN: t.owner \neq a$
Busy(*a*:*AGENT*): *BOOLEAN* $=_{\text{def}}$ $a.mode \neq undefined \wedge \exists t \in TOKEN: t.owner = a$
Available(*t*:*TOKEN*): *BOOLEAN* $=_{\text{def}}$ $t.owner = undefined$

Integrity Constraints

constraint $\forall a \in AGENT: (a.Stop \Rightarrow a.Busy)$

Initial Constraints

initially $\forall a \in$ *AGENT*: *a.program* = RESOURCE-MANAGEMENT-PROGRAM
initially $\forall a \in$ *AGENT*: *a.Idle* $\wedge \forall t \in$ *TOKEN*: *t.Available*

ASM Programs

RESOURCE-MANAGEMENT-PROGRAM:

```
do in-parallel
    SHAREDACCESS
    EXCLUSIVEACCESS
    RELEASEACCESS
enddo
where
    SHAREDACCESS ≡
        if Self.mode = shared ∧ Self.Waiting then
            choose t: t ∈ TOKEN ∧ t.Available
                t.owner := Self
    EXCLUSIVEACCESS ≡
        if Self.mode = exclusive ∧ ∀ t ∈ TOKEN: t.Available then
            do forall t: t ∈ TOKEN
                t.owner := Self
    RELEASEACCESS ≡
        if Self.Busy ∧ Self.Stop then
            Self.mode := undefined
            do forall t: t ∈ TOKEN ∧ t.owner = Self
                t.owner := undefined
endwhere
```

2.1.6 Predefined Names

To define an ASM model, in particular the ASM model capturing the semantics of RSDL, certain names and their intended interpretation are predefined. These names are grouped and listed in this section (where *D* refers to the syntactic category of domains). For prefix, infix and postfix operators an underline ("_") is used to indicate the position of their arguments. Moreover, the precedence of the operators is indicated by prec(n), where n is a number. Higher numbers mean tighter binding. Monadic operators have a tighter binding than binary ones. Binary operators are associative to the left.

ASM-specific Domains	
static domain *X*	ASM base set, only a meta domain
static domain *BOOLEAN*	Boolean values
static domain *INT*	Integer values
static domain *REAL*	Real values
shared domain *AGENT*	ASM agents
static domain *PROGRAM*	ASM programs
static domain *TOKEN*	Syntax tokens (character strings)
_ *	Domain constructor: sequence of
_ +	Domain constructor: non-empty sequence of
_ **-set**	Domain constructor: set of
_ × _ prec(7)	Tuple domain constructor
_ ∪ _ prec(6)	Union domain constructor
{ }	Enumeration domain constructor, comma-separated list of elements
_ → _ prec(8)	Mapping domain constructor

ASM-specific Functions	
static *undefined*: → *X*	Indicator for undefined values
monitored *Self*: → *AGENT*	Self reference for ASM agents
controlled *program*: *AGENT* → *PROGRAM*	Program of an ASM agent
monitored *currentTime*: → *REAL*	The current system time.
monitored *clock*: *REAL* → *REAL*	Time adjustment function
now: *REAL* $=_{def}$ *clock*(*currentTime*)	Current system time.

Boolean Functions and Predicates	
static *True*: → *BOOLEAN*	Predefined literal.
static *False*: → *BOOLEAN*	Predefined literal
_ = _ prec(4)	Equality
_ ≠ _ prec(4)	Inequality
_ ∧ _ prec(3)	Logical and
_ ∨ _ prec(2)	Logical or
_ ⇒ _ prec(1)	Implication
_ ⇔ _ prec(1)	Logical equivalence
¬ _	Negation
∃*x* ∈ *D*: *P*(*x*) prec(0)	Existential quantification
∀*x* ∈ *D*: *P*(*x*) prec(0)	Universal quantification

Terms	
x	0-ary function application
{ _ → _ } prec(8)	Mapping element
f(*t*$_1$,..., *t*$_n$)	Function application with n argument expressions
if *Formula* **then** *Term* **else** *Term* [**endif**]	Conditional expression; again we use **elseif** instead of **else if**
s-_(_)	Tuple selection function (see Tuples below)
mk-_(...)	Tuple construction (see Tuples below)
inv-_(...)	The inverse of a function or map, **inv-***Fun*(*x*) =$_{\text{def}}$ { *a* ∈ *D*: *Fun*(*a*) =*x* }.*take*

Functions and Relations on Numbers	
_ > _, _ ≥ _, _ < _ , _ ≤ _ prec(4)	Comparison operators
_ + _ prec(6)	Addition
_ - _ prec(6)	Subtraction
_ * _ prec(7)	Multiplication
_ / _ prec(7)	Division
_ **mod** _ prec(7)	Integer division
_ **div** _ prec(7)	Integer division remainder
- _	Unary minus
0, *1*, ...	Integer literals

Functions on Sequences	
static *empty*: → *D**	Empty sequence
static *head*: *D** → *D*	Head of the sequence (*undefined* when empty)
static *tail*: *D* → *D**	Tail of the sequence (*undefined* when empty)
static *last*: *D** → *D*	Last element of a sequence (*undefined* when empty)
static *length*: *D** → *INT*	Length of a sequence
static < >: *D*^{n} → *D**	Sequence constructor; arguments are listed inside the brackets, separated by commas
_ ⌢ _ prec(6)	Concatenation of sequences
toSet: *D** → *D*-**set**	Conversion of the elements of a sequence into a set.
_ [_]	Access an element of a list. The index within the brackets must be of type *INT*.
< <*result*> | <*var*> **in** <*seq*> : <*cond*> >	Sequence comprehension; acts as filter on <seq>, i.e. order-preserving
< <*var*> **in** <*seq*> : <*cond*> > =$_{\text{def}}$ < <*var*> | <*var*> **in** <*seq*> : <*cond*> >	Abbreviated sequence comprehension
< <*result*> | <*var*> **in** <*seq*> > =$_{\text{def}}$ < <*result*> | <*var*> **in** <*seq*> : *True* >	Abbreviated sequence comprehension

Functions on Sets	
_ ∪ _ prec(6)	Set union
_ ∩ _ prec(7)	Intersection
_ \ _ prec(6)	Set subtraction
_ ∈ _ prec(4)	Element of?
_ ∉ _ prec(4)	Not element of?

Functions on Sets (continued)	
_ ⊆ _ prec(4)	Subset of?
_ ⊂ _ prec(4)	Proper subset of?
| _ |	Size of a set
U _	Big union: union of all sets within the argument set
∅	Empty set
static { }: $D^n \rightarrow D$-**set**	Set constructor; comma-separated list of arguments in the brackets
take: *D*-**set** → *D*	Select an arbitrary element from the set, or *undefined* for an empty set
_ .. _ prec(5)	Integer range from the first value to the second. Empty set when the second expression is smaller than the first one.
{ <*result*> | <*var*> ∈ <*set*> : <*cond*> }	Set comprehension, acts like a filter on <set>
{ <*var*> ∈ <*set*> : <*cond*> } =$_{def}$ { <*var*> | <*var*> ∈ <*set*>: <*cond*> }	Abbreviated set comprehension
{ <*result*> | <*var*> ∈ <*set*> } =$_{def}$ { <*result*> | <*var*> ∈ <*set*>: *True* }	Abbreviated set comprehension

Patterns and Case-expressions
Patterns provide a means to easily access the structure of values. The following patterns are provided.

- Variables: A variable matches any value. However, if the variable is already bound, it only matches itself.
- Anonymous variables: Anonymous variables are denoted by "*". They are a shorthand for introducing an unused variable.
- Constructor: A constructor is given by its name and the arguments, that are again patterns. It matches any value that is constructed using that constructor and with the arguments matching their corresponding pattern.
- Named Pattern: The notation Variable = Pattern introduces a name for (the value matching) the pattern.

Patterns are used to describe functions on the syntax tree. The non-terminal names of the grammar are used as the constructor functions.
A case expression is used to determine a value depending on pattern matching.

CaseExpression ::= **case** *Term* **of**
| *Pattern*$_1$: *Term*$_1$
| *Pattern*$_2$: *Term*$_2$
...
[**otherwise** *Term*$_0$]
endcase

If the value of *Term* matches at least one *Pattern*$_i$, then the result of the case expression is given by the *Term*$_i$. If no pattern matches, the result is *Term*$_0$ (if present). Otherwise, the result is *undefined*.

Union Domains
Union domains simply contain the values of their constituent domains.

$Dom =_{def} Dom_1 \cup Dom_2$

However, sometimes we want to make clear to which sub-domain a value belongs. Therefore, the following derived functions are implicitly defined.

selectKind-*Dom*$_1$: *Dom* → *Dom*$_1$
selectKind-*Dom*$_2$: *Dom* → *Dom*$_2$
selector-*Dom*$_1$: *Dom*$_1$ → *Dom*
selector-*Dom*$_2$: *Dom*$_2$ → *Dom*

selectKind-*Dom*$_1$(*x*: *Dom*): *Dom*$_1$ =$_{def}$ **if** *x* ∈ *Dom*$_1$ **then** *x* **else** *undefined* **endif**
selectKind-*Dom*$_2$(*x*: *Dom*): *Dom*$_2$ =$_{def}$ **if** *x* ∈ *Dom*$_2$ **then** *x* **else** *undefined* **endif**
selector-*Dom*$_1$(*x*: *Dom*$_1$): *Dom* =$_{def}$ *x*
selector-*Dom*$_2$(*x*: *Dom*$_2$): *Dom* =$_{def}$ *x*

Tuples
For every declared tuple domain a number of implied constructor and selector functions is defined. A definition

$$Dom =_{\text{def}} Dom_1 \times Dom_2^* \times Dom_3\text{-}\mathbf{set} \times Dom_1 \times (Dom_1 \cup Dom_2)$$

also defines the following functions.

$\mathbf{mk}\text{-}Dom$: $Dom_1 \times Dom_2^* \times Dom_3\text{-}\mathbf{set} \times Dom_1 \times (Dom_1 \cup Dom_2) \to Dom$
$\mathbf{s}\text{-}Dom_1$: $Dom \to Dom_1$
$\mathbf{s}\text{-}Dom_2\text{-}\mathbf{seq}$: $Dom \to Dom_2^*$
$\mathbf{s}\text{-}Dom_3\text{-}\mathbf{set}$: $Dom \to Dom_3\text{-}\mathbf{set}$
$\mathbf{s2}\text{-}Dom_1$: $Dom \to Dom_1$
s-implicit: $Dom \to (Dom_1 \cup Dom_2)$

When the tuple includes the same domain more than once, selector functions similar to $\mathbf{s2}\text{-}Dom_1$ are defined. For union the special selector function **s-implicit** is defined.

Abstract Syntax Rules
Abstract syntax rules from the language definition are directly translated to the ASM notation, using certain conventions that will be explained by examples. Basically, an abstract syntax rule can be understood as declaring one or more (tuple) domains, and defining functions to construct and select values of the component domains. However, syntax nodes have an identity as opposed to ordinary tuples. There are syntax rules introducing named constructors as well as named and unnamed unions. Rules introducing constructors are composed of terminal and non-terminal symbols, they have the form

$$Symbol :: Symbol_1\ \ Symbol_2^+\ \ Symbol_3\text{-}\mathbf{set}\ \ [Symbol_4]$$

which is translated to

$Symbol\text{-}aux =_{\text{def}} Symbol_1 \times Symbol_2^* \times Symbol_3\text{-}\mathbf{set} \times Symbol_4$
controlled domain *Symbol*
controlled *contents-Symbol*: $Symbol \to Symbol\text{-}aux$
$\mathbf{s}\text{-}Symbol_1(x\text{: }Symbol)\text{: }Symbol_1 =_{\text{def}} \mathbf{s}\text{-}Symbol_1(x.contents\text{-}Symbol)$
$\mathbf{s}\text{-}Symbol_2\text{-}\mathbf{seq}(x\text{: }Symbol)\text{: }Symbol_2^* =_{\text{def}} \mathbf{s}\text{-}Symbol_2\text{-}\mathbf{seq}(x.contents\text{-}Symbol)$
$\mathbf{s}\text{-}Symbol_3\text{-}\mathbf{set}(x\text{: }Symbol)\text{: }Symbol_3\text{-}\mathbf{set} =_{\text{def}} \mathbf{s}\text{-}Symbol_3\text{-}\mathbf{set}(x.contents\text{-}Symbol)$
$\mathbf{s}\text{-}Symbol_4(x\text{: }Symbol)\text{: }Symbol_4 =_{\text{def}} \mathbf{s}\text{-}Symbol_4(x.contents\text{-}Symbol)$

Moreover, there is an abbreviation $\mathbf{mk}\text{-}Symbol$. This abbreviation amounts to creating a new object of domain *Symbol* using the **extend** primitive and to set the *contents-Symbol* value of the newly produced object to the result of $\mathbf{mk}\text{-}Symbol\text{-}aux$. Note that this kind of abbreviation is not a function, but in fact a rule item. Therefore, it must be used only within rules. The fact that the optional $Symbol_4$ is not present is expressed in the ASM model by leaving the corresponding value *undefined*.
An empty sequence of symbols (constructor with no parts) is denoted by ( ).
The equality for syntax values is always a structural equality, i.e. the contents of the symbols are compared instead of the symbols themselves.
The syntax rules introducing named unions, i.e., synonyms, have the form

$$Symbol = Symbol_1 \mid Symbol_2 \mid \ldots \mid Symbol_N \qquad (n \geq 1)$$

which is translated to

$$Symbol =_{\text{def}} Symbol_1 \cup Symbol_2 \cup \ldots \cup Symbol_n$$

Note that since *Symbol* is a *union* domain, the expansion yields a domain definition, but no functions **mk-** or **s-**.

In some cases, it is not necessary to refer to synonyms. Here, unnamed unions may be introduced by

$Symbol :: Symbol_1 \{ Symbol_{21} | ... | Symbol_{2N} \}$

instead of introducing synonyms.

$Symbol :: Symbol_1\ Symbol_2$
$Symbol_2 = Symbol_{21} | ... | Symbol_{2N}$

For each RSDL keyword **KEYWORD**, there is an associated keyword domain *Keyword* with just one value:

static domain *Keyword*

It is required that all keyword domains are mutually disjoint.
Given the abstract grammar, there is a derived domain called *DefinitionAS1*, which is composed of all abstract syntax symbol domains as follows:

$DefinitionAS1 =_{def} Symbol_1 \cup Symbol_2 \cup ... \cup Symbol_n$

where $Symbol_1$,$Symbol_2$,...,$Symbol_N$ is the list of *all* terminal and non-terminal symbols of the abstract grammar.
There is a similar domain *DefinitionAS0* for the concrete grammar (AS0).
To navigate in a given abstract syntax tree, the functions **s-** can be used to find the children and two parent functions are defined below to find the parents.

controlled *parentAS1*: *DefinitionAS1* → *DefinitionAS1*
controlled *parentAS0*: *DefinitionAS0* → *DefinitionAS0*

Moreover, two functions are defined to find the parent of a particular kind.

parentAS0ofKind(*from*: *DefinitionAS0*, *x*: *DefinitionAS0*-**set**): *DefinitionAS0* =$_{def}$
  if *from* = *undefined* **then** *undefined*
  elseif *from* ∈ *x* **then** *from*
  else *parentAS0ofKind*(*from.parentAS0*, *x*)
parentAS1ofKind(*from*: *DefinitionAS1*, *x*: *DefinitionAS1*-**set**): *DefinitionAS1* =$_{def}$
  if *from* = *undefined* **then** *undefined*
  elseif *from* ∈ *x* **then** *from*
  else *parentAS1ofKind*(*from.parentAS1*, *x*)

The root node of the current abstract or concrete syntax tree is denoted by the following nullary functions.

controlled *rootNodeAS1*: → *DefinitionAS1*
controlled *rootNodeAS0*: → *DefinitionAS0*

2.2 Implementation Technology

In order to make an automatic generation of an implementation of the formal RSDL semantics feasible it is necessary to use meta tools. This means to use tools that in turn generate other tools, e.g. compiler tools.
We use the tools `lex` (`flex`) and `yacc` (`bison`) from the usual UNIX distributions. These two care for the handling of the language syntax. Moreover it is necessary to care for the semantics. This is done here with the tool `kimwitu`. Using `kimwitu` it is possible to define an abstract syntax tree and to define operations on it on a quite high level.
Please find an overview of the methodology in Figure 7 below. In the centre of the methodology there is the abstract syntax tree, which is defined using the meta tool `kimwitu`. The abstract syntax tree is just another representation of the input, but it is abstracted such that purely syntactical information is not included here. So the abstract syntax tree resembles quite exactly the structure and information of the input, it will just omit unimportant details. It would even be possible to automatically generate the abstract syntax from the concrete syntax as done in Section 5.3.4, but usually it will be more effective to do this manually.

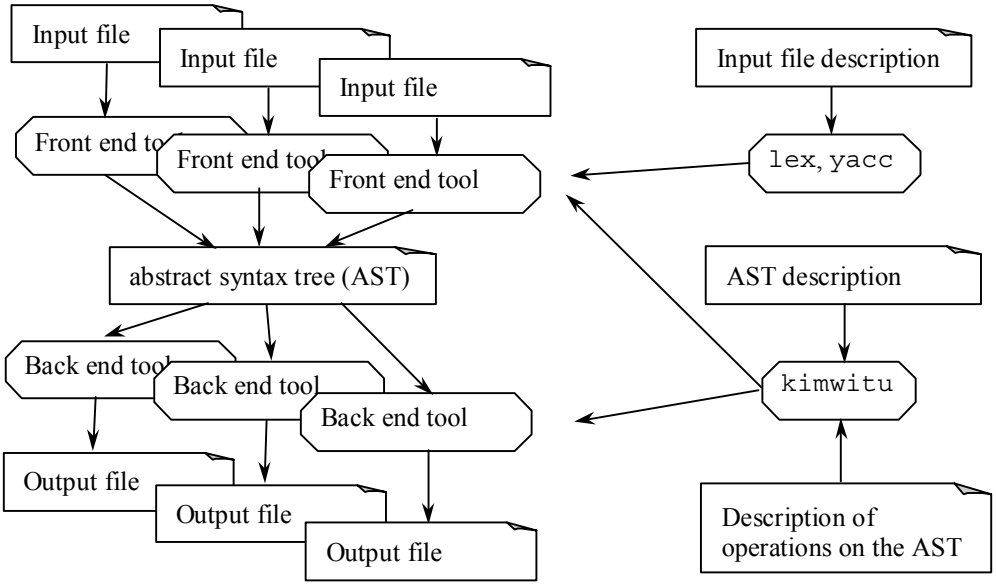


Figure 7: Methodology overview

The concrete format of the input is described using the meta tools `lex` for the lexical rules of the input format and `yacc` for the syntax rules. In fact, in the scope of Linux we are using the variants `flex` (instead of `lex`) and `bison` (instead of `yacc`) of these tools. These tools are used such that they generate the abstract format as defined with `kimwitu`. In Figure 7 one can see that the front end tools are generated using `lex` and `yacc` and the abstract syntax tree construction is generated by `kimwitu`. The interworking of these three goes like this: `yacc` is in the centre of the three, it uses a function from `lex` to identify lexical tokens, and then uses tree construction functions from `kimwitu` to construct the abstract syntax tree. Usually there will be only one input format, but it is also possible to have many of then, even to handle generations of input formats. The RSDL semantics implementation will use one input format for the ASM part and three input formats for the various kinds of grammars. The abstract syntax must, however, be the same when many input formats are used. It is possible not to use all of the possibilities of the abstract syntax for every input format.
The definition of the back end tools is then rather straightforward. Starting from the abstract syntax tree various outputs may be generated. The output generation is always described in terms of the abstract syntax tree and will also follow the structure of the abstract syntax tree. `Kimwitu` provides the means to describe what to generate from the abstract syntax tree. Moreover, it provides means to describe necessary transformations over the abstract syntax tree that have to be done before the transformation takes place. This is used in several places in the example.
In order to gain a better understanding for the tools you should consult Section 2.3, where an example is used to explain the meta tools mentioned above.

2.2.1 Abstract Syntax Trees

An abstract syntax tree first of all resembles an ordinary tree. The nodes of the tree are typed according to the abstract syntax. There are only two kinds of nodes, namely alternative nodes and sequence nodes. Sequence nodes are built using a constructor. They may have other nodes in them (children). Alternative nodes are just an alternative of sequence nodes. Although the grammar of RSDL is described in general BNF it is possible to transform it into an abstract tree format using auxiliary node types.

2.2.2 Front End Tools

All the front end tools have the task to transform a concrete grammar into an abstract syntax tree. To this end, the tools `lex` and `yacc` are used for the generation of the front end tools. The lexical structure is analysed using `lex`, which provides lexical tokens. These tokens are used by `yacc` to parse the syntax. `Yacc` will in turn use the tree construction functions provided by `kimwitu` to build an abstract tree according to the parse results.

2.2.3 Back End Tools

There are two kinds of backend tools: tools that do transformations and/or checks on the abstract grammar, and tools that generate output from the abstract grammar. Both kinds are realised using `kimwitu` features, namely unparse rules for generation of output and rewrite rules for transformations.

2.3 Implementation Meta Tools

This section explains the tools used for implementation of the formal semantics definition of RSDL. Please note, that you will not find an in-depth explanation of the tools here. For any of the tools, it is only explained as much as is necessary to make the methodology work. Please look into section 6.1 for references to more material.

2.3.1 Kimwitu

The tool `kimwitu` is designed to enable easy handling of abstract syntax trees. This accounts for several things to be possible. `Kimwitu` allows to

- define the structure of abstract syntax trees,
- define functions over the abstract syntax tree,
- define output to be generated along the structure of the abstract syntax tree, and to
- define rewriting of the abstract syntax tree in order to achieve a normal form of the tree.

There is an input format of `kimwitu` that allows to do all of the above. `Kimwitu` then checks whether the input is correct according to its rules and produces as output C functions that provide the functionality wanted.

2.3.1.1 Defining a Syntax Tree

`Kimwitu` allows just for very few constructs to define an abstract syntax tree. Every node within the tree has a node type (which is also called a phylum) and it is constructed using a constructor. The constructor can put together nodes, which are then the children of that node. So basically a `kimwitu` abstract grammar definition is a declaration of the constructors and their corresponding node types.
Let us assume we want to represent an EBNF (extended BNF) syntax. There we distinguish terminal symbols and syntax rules (non-terminals). A terminal symbol is just a name and could be declared as follows.

```
rule: Token( casestring ) ;
```

Please note, that `casestring` is the built-in node type for character string tree nodes (leaf node). Its constructor is called `mkcasestring`.
A non-terminal is characterised by a syntax rule, so we could represent it in `kimwitu` as

```
rule: Rule( casestring expression ) ;
```

Alternatively, we could have written these two definitions together like

```
rule:
    Rule( casestring expression )
  | Token( casestring )
  ;
```

The whole syntax is characterised by a sequence of such rules describing all node types and all constructors. `Kimwitu` provides a special list construct to denote sequences.

```
syntax: list rule;
```

This declaration is almost the same as a the following declaration.

```
syntax: Nilsyntax() | Conssyntax( rule syntax ) ;
```

The difference is that `kimwitu` will automatically generate some useful functions for lists when they are declared using the list construct, as e.g. length, concatenation, etc.
Now we go on with the declaration of the abstract syntax of the EBNF grammar. We have to define what expressions are. Expressions are simply sequences of alternatives. We introduce a node type `serial` for the alternatives. But what is a `serial`? It is just a sequence of `atom`s. Finally, `atom`s are terminals or non-terminals or constructs of atoms, namely repetitions (1..n) or (0..n), optional parts or an expression within parentheses.
This is expressed with the `kimwitu` declarations below. Comments are written within `/*` and `*/`.

```
expression: list serial;
serial: list atom;
atom: Terminal( casestring )
  | Nonterminal( casestring )
  | AnyAtom( atom )       /* arbitrary repetition */
  | NonZeroAtom( atom )   /* repetition at least once */
  | ZeroOneExpression( expression ) /* optional parts */
  | SubExpression( expression ) /* parenthesised expression */
;
```

Now that we have defined the abstract syntax for grammars, we can proceed working with it. Kimwitu generates some C functions from this declaration of the abstract syntax tree. For our purposes it is important, that there are constructor functions corresponding to the constructors and types corresponding to the node types. These types and constructors can be used within C programs. For instances it would be valid to write something like the following.

```
expression localvar = SubExpresssion(Nilexpression());
```

In the C fragment above, the type `expression` as well as the functions `SubExpression` and `Nilexpression` are generated by kimwitu.

2.3.1.2 Unparsing the Tree

The next step is to use the abstract tree representation. Suppose we have some grammar represented as an abstract tree according to the declarations above. We want to print it out in some standard format (pretty printing). `Kimwitu` provides so-called unparse rules for this purpose. These rules have a pattern that defines their applicability and a body defining what to do if the pattern matches. If there is more than one matching pattern then the most special one is taken. The rules are also grouped according to their purpose when we want to define several sets of unparse rules for several purposes: pretty printing, C-code generation, static semantic checking etc. This is expressed in kimwitu using unparse view names. Each view name identifies a set of unparse rules. An unparsing rule looks like

`<pattern>: [ <uview>: <unparse sequence> ];`

Hereby <pattern> denotes a pattern, <uview> is the name of an unparse view and <unparse sequence> is the description of what to generate. The <unparse sequence> may contain ordinary strings, variable names (occurring in the pattern) or arbitrary C code enclosed within curly braces(“{“ and “}”). The strings are printed as they are, the variables are tree nodes and they are recursively unparsed according to the unparse rules. The C-code is transformed verbatim to the generated output.

So lets define the rules to pretty print the grammar. There is a difficulty involved in this unparsing, as expressions should be handled differently when they appear within a sub expression or when they appear in a top level rule. On top level, we want alternatives to appear on separate lines whereas otherwise alternatives should be on the same line. We introduce a C variable `level` to handle this distinction.

```
1.  /* Pretty print rules */
2.  %{ KC_UNPARSE
3.  int level = 0;
4.  %}
5.  %uview pretty;

6.  Conssyntax( head, tail )
7.  -> [pretty: tail "\n" head ];

8.  Rule( name, Consexpression( head, Nilexpression() ))
9.  -> [pretty:  name " ::=\n\t" head ".\n" ];

10. Rule( name, expr )
11. -> [pretty:  name " ::=\n" expr "\t.\n" ];

12. Token( name )
13. -> [pretty: "token(" name ")." ];

14. Consexpression( head, Nilexpression() )
15. -> [pretty: { if (!level) } "\t  "
16.  head
17.  { if (!level) } "\n"
18. ];

19. Consexpression( head, tail )
20. -> [pretty: tail
21.  { if (level) } " | " { else } "\t| "
22.  head
23.  { if (level) } " " { else } "\n"
24. ];

25. Consserial( head, Nilserial() )
26. -> [pretty: head ];

27. Consserial( head, tail )
28. -> [pretty: head
29.  { if (!level) } "\n\t  " { else } " "
30.  tail
31. ];
```

```
32. AnyAtom( a )
33. -> [pretty: "{ " a " }" ];

34. NonZeroAtom( a )
35. -> [pretty: "{ " a " }+" ];

36. ZeroOneExpression( expr )
37. -> [pretty: { level++; } "[ " expr " ]" { level--; } ];

38. SubExpression( expr )
39. -> [pretty: { level++; } "( " expr " )" { level--; } ];
```

Lines 2-4: C declarations for the unparsing: the variable `level` is declared.
Line 5: declaration of an unparsing view `pretty`
Lines 8-11: these two rules describe the handling of the `Rule` constructor. The first one is more special, so it will be applied when the rule has only one alternative. In words, the first rule says: print the `name`, then "`::=`" and "`\n\t`" (newline and tabulator) then the alternative `head` followed by "`.`" and "`\n`". The unparsing of `head` is performed according to the rules on lines 25-31.
Lines 14-24: The C-notation assures that the "`\t`" and "`\n`" are printed only when `level==0`. Please note, that usually lists with only one element need a special handling. You might have noticed that all lists are processed from tail to head. This is due to their construction: they are also constructed from tail to head. See Section 2.3.2 in this respect.
Lines 36-39: These lines show how the top level expression handling is switched off for nested expressions: the level is increased before the inner expression and decreased afterwards.

2.3.1.3 Rewriting the Tree

Another thing that we want to do with the tree is to transform it. Some constructs contain irrelevant details that could be thrown away. One example is when we have a sub expression that itself contains a single sub expression. This could be simplified with the rewrite rule below.

```
SubExpression(Consexpression(Consserial(Subexpression(s),Nilserial()),Nilexpression()))
-> < basic_rewrite: Subexpression(s) >;
```

Rewrite rules are also based on patterns, and define for those patterns new patterns that are to be inserted into the place of the old pattern. A rewrite rule looks as follows.
`<pattern>: < <rview>: <value> >;`
Hereby <rview> denotes a rewrite view. As for the unparsing rules, a rewrite view groups together rewrite rules that belong together.
Please look below for the definition of some more simplifying rules for the grammar example.

```
%rview basic_rewrite;

/* this first rule simplifies subexpressions within subexpressions */
SubExpression(Consexpression(Consserial(Subexpression(s),Nilserial()),Nilexpression()))
-> < basic_rewrite: Subexpression(s) >;

/* The following two rules simplify repetitions with subexpression in them */
AnyAtom(
  SubExpression(Consexpression(Consserial(a,Nilserial()),Nilexpression())))
-> < basic_rewrite: AnyAtom(a) >;

NonZeroAtom(
  SubExpression(Consexpression(Consserial(a,Nilserial()),Nilexpression())))
-> < basic_rewrite: NonZeroAtom(a) >;

/* If the rule is empty, only a Nilexpression is used */
Rule(N, Consexpression(Nilserial(), Nilexpression()))
-> < basic_rewrite: Rule(N, Nilexpression() ) >;
```

This is a good place to say something more about patterns. Patterns are recursively defined as follows.
- A variable is a pattern.
- An anonymous variable ("*") is a pattern.
- A constructor name, optionally followed by a parenthesised list of patterns is a pattern. There must be the same number of patterns as there are arguments in the constructor declaration.
- An equation <variable> = <pattern> is a pattern.

Nothing more is allowed for patterns. To describe the resulting <value> of a rewrite rule, we can use
- the variables from the pattern,
- the tree construction functions, and
- arbitrary C functions.

2.3.1.4 Symbol Tables

Now we want to investigate the kimwitu possibilities to define symbol tables. First, we define a node kind for symbol tables and two unparse views.

```
%view create_symtab, check_symtab;

symtab: list symbol;
symbol {uniq}: NT(casestring) | TT(casestring)
         { int defined = 0; int used = 0; };
```

Please note first the annotation `uniq`. This means, that the symbol nodes are only newly created, when they are really new. If already one node with the same children exists, only a reference to this node is produced and no new node created. An example for this behaviour is the phylum `casestring`.
The declaration of `defined` and `used` introduces C variables that are attached to the `symbol` nodes. They will serve for counting of definitions and uses afterwards.
The next step is to introduce various C declarations. All lines included within `%{` and `%}` are transferred to the generated C files. An optional redirection name may identify the destination file. No redirection name means the current C file, `HEADER` means the current header file and `KC_UNPARSE` means the unparsing C file. `KC_REWRITE` stands for the rewriting C file and `KC_TYPES_HEADER` means inclusion into every generated file.

```
%{
#include <stdio.h>
symtab TheSymtab;
%}

%{ HEADER
#include "unpk.h"
%}

%{ KC_UNPARSE
#include <ctype.h>
#include "ebnf-semantics.h"

static int errorcount=0;

void error(char *s, char *p)
{ fprintf(stderr, "error: %s%s\n",s,p); errorcount++; }

void warning(char *s, char *p)
{ fprintf(stderr, "warning: %s%s\n",s,p); }
%}

%{ KC_REWRITE
#include "ebnf-semantics.h"

extern symtab TheSymtab;
%}
```

Now the first step of the static analysis starts: construction of the symbol table. First functions to create symbol table entries are defined:

```
void init_symtab() { TheSymtab = Nilsymtab(); }

void insert_nt(casestring name, int def, int use)
{ symbol sym = NT( name );
  if ( sym->defined + sym->used == 0 ) TheSymtab = Conssymtab( sym, TheSymtab );
  sym->defined+= def; sym->used+= use;
}

void insert_tt(casestring name, int def, int use)
{ symbol sym = TT( name );
  if ( sym->defined + sym->used == 0 ) TheSymtab = Conssymtab( sym, TheSymtab );
  sym->defined+= def; sym->used+= use;
}
```

Please note the use of the local variable sym. When `sym->defined + sym->used == 0` then the symbol is newly created and has to be inserted into the global symbol table list `TheSymtab`.
Now we define the unparsing rules for the symbol table creation. The first rule is merely to ensure that the error messages appear in the correct order.

```
Conssyntax(h, t)  -> [ create_symtab: t h ];

Rule(name, e)     -> [ create_symtab: { insert_nt( name, 1, 0 ); } e ];
```

```
Nonterminal(name) -> [ create_symtab: { insert_nt( name, 0, 1 ); } ];

Token(name)       -> [ create_symtab: { insert_tt( name, 1, 0 ); } ];

Terminal(name)    -> [ create_symtab: { insert_tt( name, 0, 1 ); } ];
```

Now we start with checking the symbol table. This means just to go through the list of all symbol table entries and to check their local variables.

```
NT( name ) -> [ check_symtab:
       { if(!$0->defined) error("undefined nonterminal: ",name->name); }
       { if($0->defined>1) error("multiply defined nonterminal: ",name->name); }
       { if(!$0->used) warning("unused nonterminal: ",name->name); }  ];
TT( name ) -> [ check_symtab:
       { if(!$0->defined) error("undefined terminal: ",name->name); }
       { if($0->defined>1) error("multiply defined terminal: ",name->name); }
       { if(!$0->used) warning("unused terminal: ",name->name); } ];
```

Please note the use of `$0` as a reference to the left hand side of the rule and the access to the node variables using the `->` notation. For the phylum `casestring`, there is a node variable `name` referring to the C string of that node.

2.3.2 Yacc

The second step after the definition of the abstract grammar of the language (which is EBNF in our case) is to construct a parser that will generate such an abstract tree from a concrete specification/program of the language. `Yacc` (yet another compiler compiler) is a tool that enables automatic generation of a parser. A parser looks through the input file in order to find out if the input structure is correct. The parser works on a sequence of tokens. This token sequence could be the sequence of characters of the program or some collated sequence which is produced by a tool. In our case we assume some program that checks the input for larger entities as e.g. names, keywords and special symbols. The parser would only find out, if the structure of the input was right (in fact, it would find out, if it was a correct sequence of tokens). For the methodology proposed here, that would not be enough, as the aim with the parser is to produce the intermediate abstract syntax tree representation. This can be done using `yacc`, as it is possible to define actions that have to be performed when syntax rules have been used successfully. For the actions, we use the tree generation functions generated by `kimwitu`. So there is a tight relation between `yacc` and `kimwitu` in this approach. Please note, that there are some tasks usually done by a compiler that refer to the syntactical correctness of the input, but which are not related to the syntax rules. These are for instance usage of correct identifiers, definition before use rules and typing rules. These are collectively referred to as static semantic rules. They are checked after the syntax check was done, and they are defined on the abstract syntax tree. See Section 2.3.1.4 for more information.

2.3.2.1 Defining EBNF in Yacc

A `yacc` input file consists of three major parts. The first part defines the interface of `yacc` to the outside, i.e., which information `yacc` is using. The second part defines the interior of `yacc`, i.e. the syntax rules and everything that belongs to this. The third part is then devoted to the definition of auxiliary functions that are used within the second part. The parts are separated by a `%%` sign. For our EBNF example, we need the following declarations in the first part.

```
1.  %{
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include "k.h"

6.  extern void yyerror(char*); /* comes from the lexer */
7.  extern int yylex();         /* comes from the lexer */
8.  extern syntax TheSyntax;
9.  %}

10. %token                 ASSIGN
11. %token <yt_casestring> DEFNT NONTERMINAL TERMINAL TOKEN

12. %type  <yt_syntax>       syntax
13. %type  <yt_rule>         rule token
14. %type  <yt_expression>   expression
15. %type  <yt_serial>       serial
16. %type  <yt_atom>         atom

17. %%
```

Lines 2-5 include the tree construction functions as generated by `kimwitu` and the some standard libraries.
Lines 5-8 declare the external entities as provided by the lexer and the main syntax tree node.
Lines 10 and 11 define the terminal symbols used and lines 12-16 the types of the syntax constructs. As we want to construct syntax trees, the types of the nodes refer to the `kimwitu` node types as included with `k.h`. The file `k.h` does already include all definitions necessary for `yacc`, such that declaration is easy.
The interface to the scanner (`lex` in this case) is built-in into `yacc`, i.e. the use of a scanner like the one generated by `lex` is presupposed.
The main part of the `yacc` file declares the syntax rules. Please find below the rules for our example. All syntax entities, terminals as well as non-terminals, have a value associated with them (the so-called semantic value). The value for non-terminal nodes is constructed from the constituent nodes using the `kimwitu` tree construction functions. In this context, `$$` is the reference to the left hand side non-terminal of the rule and `$1`, `$2`, ... are the references to the values of the right hand side.

```
18. spec: syntax
19.       { TheSyntax = $1; }
20.     ;

21. syntax:  /* empty */
22.       { $$ = Nilsyntax(); }
23.     | syntax rule
24.       { $$ = Conssyntax( $2, $1 ); }
25.     | syntax token
26.       { $$ = Conssyntax( $2, $1 ); }
27.     ;

28. rule:
29.       DEFNT ASSIGN expression
30.       { $$ = Rule( $2, $1, $3 ); }
31.     ;

32. token:
33.       TOKEN
34.       { $$ = Token( $1 ); }
35.     ;

36. expression: /*empty*/
37.       { $$ = Nilexpression(); }
38.     | serial
39.       { $$ = Consexpression( $1, Nilexpression() ); }
40.     | expression '|' serial
41.       { $$ = Consexpression( $3, $1 ); }
42.     ;

43. serial:
44.       atom
45.       { $$ = Consserial( $1, Nilserial() ); }
46.     | atom serial
47.       { $$ = Consserial( $1, $2 ); }
48.     ;

49. atom:
50.       TERMINAL
51.       { $$ = Terminal( $1 ); }
52.     | NONTERMINAL
53.       { $$ = Nonterminal( $1 ); }
54.     | '{' expression '}'
55.       { $$ = SubExpression( $2 ); }
56.     | '[' expression ']'
57.       { $$ = ZeroOneExpression( $2 ); }
58.     | atom '*'
59.       { $$ = AnyAtom( $1 ); }
60.     | atom '+'
61.       { $$ = NonZeroAtom( $1 ); }
62. ;

63. %%
```

The following points are worth noting.
Lines 18-20: The first rule is the main rule. It stores the semantic value into the external variable `TheSyntax`.
Lines 21-27: If the syntax is empty, then the semantic value is `Nilsyntax()`. It may also be a syntax followed by a rule or by a token declaration. In this case, the new element is put in front of the list. Here you see how the list is built upside-down: the last element is put into the front.

Lines 28-31: In order to cope with the improper line endings in the EBNF grammar a special terminal kind `DEFNT` is introduced. It refers to a non-terminal name before a "`::=`". The token construction of the scanner has to care for this.
lines 43-48: Here we have another case: the tree construction is in the correct order. Usually it is better in `yacc` to use rules that are left recursive like the rule for `expression`. However, sometimes it is necessary to use right recursive rules as here. It is important to handle this properly in terms of the `kimwitu` tree construction.
The last part of the `yacc` file is empty in this example.

2.3.2.2 Syntax Rules

The input language of `yacc` is somewhat sparse when it comes to syntax definitions. However, all features for context free grammars can be defined. The input language is a variant of BNF. A syntax rule has a left-hand-side, this is the non-terminal to be defined. There is also a right-hand-side, which is the syntax rule for this non terminal. The right hand side consists of a several alternatives which can be used to construct this non terminal. Any of those alternatives is a sequence of syntax elements, either terminals or non terminals. Such a sequence could also be empty. These elements are all that is allowed in `yacc`.
`Yacc` will attempt to produce a parser from the syntax rules. It will however not always be possible to do so, as the language has to be LALR(1) in order for `yacc` to work properly.[9] Sometimes there are ambiguities in the grammar that could be avoided by `yacc` when there were more information. Therefore `yacc` provides means to define operator precedences and associativities.
`Yacc` tries to analyse the input using an LALR(1) strategy, which is based on a stack and one token look-ahead. `Yacc` in fact implements a stack automaton as follows. There is always a state `yacc` is in. Depending on the state and on the next (looked ahead) input symbol `yacc` figures out what to do next. There are three possible actions.

- *Shift* to a new state: this means to put the input symbol into the stack and to go to a new state
- *Reduce* according to a rule: this means to remove the n topmost stack items as identified by the number of elements of the right hand side of the rule and to replace then by one item as identified by the left hand side of the rule. Moreover, the semantic action (computation of the semantic value of the new item) takes place here.
- *Go to* state: this means to just enter a new state without further action.

In order for this simple setup to work, the grammar must be LALR(1), i.e. it must be possible to find out with one token look-ahead what to do next. Sometimes this is not the case, then conflicts occur.

2.3.2.3 Conflict Resolution

There are basically two kinds of conflict in `yacc`, namely *shift-reduce conflicts* and *reduce-reduce conflicts*. Information about conflicts is included in the file `y.output` which is created when `yacc` is called with the option `-v`. It is always necessary to know what exactly causes the conflict before resolution actions can be taken.
Shift-reduce conflicts mean that it is impossible to decide when a syntax construct ends. It is caused by one alternative being a proper substring of another one. The well-know problem with if-then-else belongs to this category. Often these conflicts can be solved by telling `yacc` which alternative to choose. This is done using operator precedences. By default, `yacc` uses shift (which means binding to the left).
Reduce-reduce conflicts mean that it is impossible to decide by which rule to reduce. It is caused by two overlapping alternatives. This kind of conflict is best solved by making the alternatives distinct. By default, `yacc` chooses one of the candidate alternatives (the first one).
Token precedences can be used to solve shift-reduce conflicts.
This is done by declarations of the kind

```
%left <tokenlist>
%right <tokenlist>
%nonassoc <tokenlist>
```

When a shift-reduce conflict occurs and the next token (the one possibly shifted – we call it here the shift token) occurs in a precedence declaration of the kind above, and within the symbols that would be reduced there is also a token occurring in a precedence declaration (we call such a token the reduce token), then `yacc` decides between shift and reduce as follows.

- If the shift token appears in a precedence declaration before the reduce token, then it is a reduce.
- If the shift token appears in a precedence declaration after the reduce token, then it is a shift.
- If both tokens appear in the same rule, the kind of the rule is important: `left` means reduce, `right` means shift, and `nonassoc` means error.

This gives the intended precedences: `left` binds to the left, `right` to the right and `nonassoc` disallows binding. Tokens declared later bind tighter.

---

[9] In fact, yacc will always be able to generate a parser, even from grammars that are not LALR(1). However, the generated parser will not fully match the language description in this case.

2.3.3 Lex

The third step in the handling of the EBNF language is to define its lexical structure. Recall that `yacc` works on a sequence of tokens, that have to be generated by some function.
`Lex` is a tool that enables an automatic generation of a scanner, i.e. of a tool that performs the division of a sequence of characters into a sequence of tokens.

2.3.3.1 Lexical Structure of EBNF

A `lex` input file consists of three parts. The first part is for defining the interface of `lex` to the outside world. The second part defines the lexical rules; and the third part is for defining additional functions. However, most of the functions used in the scanner will be simple and better defined using C macros, i.e. `#define`. The parts are again separated by a `%%` sign.
We start with the first part of the `lex` description.

```
1.  %option noyywrap
2.  %{
3.  #include <stdio.h>

4.  #ifdef DEBUG

5.  int main()
6.  { char *p;
7.    printf("checking lexis\n");
8.    while(p=(char*)yylex())
9.    printf("%-20.20s is <%s>\n", p, yytext);
10. }
11. #define MakeCASE
12. #define token(x) (int) #x

13. #else

14. #include "k.h"
15. #include "ebnf-parse.h"
16. #define token(x) x
17. #define MakeCASE { yylval.yt_casestring = mkcasestring(yytext); }

18. #endif DEBUG

19. static int yflineno = 1;

20. void yyerror ( s ) char *s;
21. { fprintf( stderr, "syntax error at line %d: %s\n", yflineno, s ); }
22. %}

23. NTNAME  \<[A-Za-z0-9 ]+\>
24. TERMNAME    [A-Z]+|[a-z]+|(\'.\'
25. SPACE   [\t \r]
26. FULLSPACE   [\t \n\r]

27. %x comment
28. %x tokendef
29. %%
```

Explanations:
Lines 1-20: Information about inclusion of `kimwitu` parts and the necessary declarations for outside world.
Line 5-12: The `lex` file is built to enable debugging of the lexical rules (display which lexical tokens are found).
Lines 14-17: These are declarations for the real lexer. We use again the definitions of `k.h` in order to construct token values. The file `y.tab.h` contains the definition of the `yacc` token values.
Line 19: A variable for the line number counting.
Lines 20-21: Definition of an error function that displays the error text and the current line number.
Lines 23-26: These lines define abbreviations for lexical patterns. These can be used afterwards by surrounding the name by curly brackets.
Lines 27-28: Declaration of lexer states.
The main part contains the patterns to be recognised by the lexer and an action to be performed. We have basically the following actions.

- do nothing: the pattern is skipped and the next pattern selected. This is indicated by a single "`;`" (no action).
- return basic value: a character or a simple token without semantic value are simply provided to `yacc`. This is indicated by `return <token name>;`.
- return with semantic value: first, a semantic value is constructed (using the tree construction functions) from the character sequence matching the pattern, then the appropriate token is returned as above.

- Start new lexer state: the lexical structure might have different parts that have different patterns each. An action `BEGIN(<state name>)` switches to a new lexer state.

```
30. \/\*              { BEGIN(comment); }
31. <comment>\*\/     { BEGIN(0); }
32. <comment>.        ; /* ignore characters in comments */
33. <comment><<EOF>> { yyerror("EOF in comment"); exit(1); }
34. <comment>\n       { yflineno++; }
35. ::=               { return token(ASSIGN); }
36. \n                { yflineno++; }
37. {SPACE}+          { /* ignored */ }
38. [[\]{}|+*]        { return token(yytext[0]); }
39. token\(           { BEGIN(tokendef); }
40. <tokendef>{TERMNAME} { MakeCASE; return token(TOKEN); }
41. <tokendef>\)      { BEGIN(0); }
42. {TERMNAME}        { MakeCASE; return token(TERMINAL); }
43. {NTNAME}          { MakeCASE; return token(NONTERMINAL); }
44. {NTNAME}/{FULLSPACE}*::  { MakeCASE; return token(DEFNT); }
45. .                 { MakeCASE; return token(TERMINAL); }
46. %%
```

The third part of the `lex` file is empty.

2.3.3.2 Lexical Rules

The lexical rules are defined using regular expressions for the patterns that match a particular token. It has to be defined how the tokens look like. For regular expressions the following expressive means are defined.

Any character	A character matches itself. This is not true for the special characters as defined below. These can be used when their special meaning is disabled by the escape character "\".
Escaped character	An escaped character matches itself.
Character selection [ ]	Matches any character from within the square brackets.
Character repetition *	Matches any sequence of the regular expression defined before.
Character repetition +	Matches any nonempty sequence of the regular expression defined before.
Grouping ()	Parentheses are used to group lexical regular expressions together.

2.3.4 Make

The final step is to put all the programs together. We already had the tools `kimwitu`, `yacc` and `lex` and we also need a C-compiler to produce executables of our specification. However, it is not easy to find out which tool to call in order to get the appropriate output. Figure 8 below depicts the dependencies that we have already with the files mentioned in the previous sections. Please note that `kc` is the name of the `kimwitu` tool.

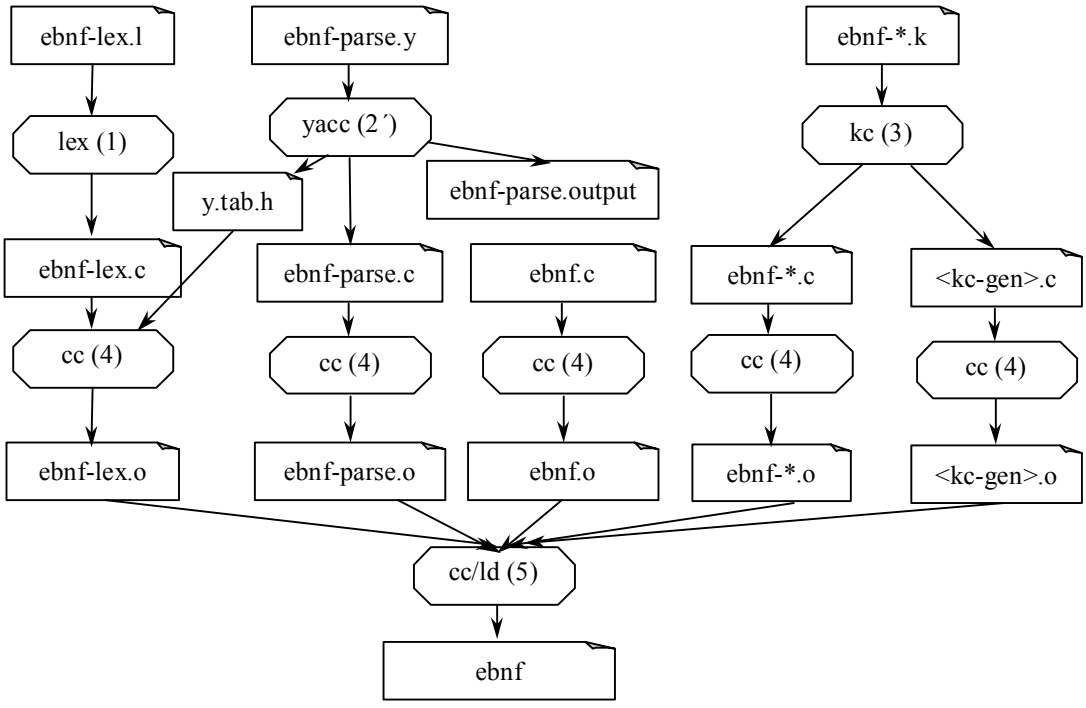


Figure 8: File Dependencies

Please note that this simple example does not fully follow the methodology as introduced in Section 2.2 because front end and back end are merged into one tool.
The program `make` is designed to ease the task of defining dependencies between programs. The idea is to simply state what the dependencies are and `make` will itself care for all of them. In our case, some of the dependencies are already built-in into `make`. `make` does already know how to create a `.c`-file provided it has a `.y` file (it uses `yacc` then) or how to create a `.c` file from a `.l` file. Other dependencies have to be given explicitely.
In Figure 8 the different kinds of dependencies that have to be handled by make are numbered. They are implemented as follows: (1), (2) and (4) are already built-in into make. It is only necessary to state what the concrete names are for `lex`, `yacc` and `cc` and which options to use for them. We also define a variable `WINNT` to indicate if the `makefile` is used under cygnus and a variable `SHELL` to denote the standard shell.

```
WINNT=1
# Tools
ifndef WINNT
SHELL   = /bin/sh
endif
YACC    = bison #-v
CC      = g++ -g
KC      = kc++
LEX     = flex

ifdef WINNT
EXE     = .exe#     # postfix for executables
endif

# Flags
YFLAGS  = -d -y
LFLAGS  = -t
CFLAGS  = -Wall -DYYDEBUG -DYYERROR_VERBOSE
```

Furthermore, we introduce abbreviations for the `lex`, `yacc` and `kimwitu` input files and for the final program.

```
# Sources
KFILES  = ebnf-abstract.k ebnf-semantics.k ebnf-trans.k ebnf-pretty.k
YFILE   = ebnf-parse.y
LFILE   = ebnf-lex.l

EBNF    = ebnf${EXE}
```

Now we start with the first real `make` rule. The first rule is special, because `make` tries to build the object that is on its left hand side when called without arguments. We do only insert a dummy target here to print out the correct usage.

```
# default rule
notknown: ; @echo "try make all"

all: ${EBNF}
```

So it remains to formulate the dependencies (3) and (5). `Kimwitu` is a really intelligent tool. It will not change its output files when they would be generated the same way as they already are. This is good when the programs afterwards are concerned, because the `cc` input files are not changed and hence `cc` will not be called. However, this way the output files of `kimwitu` could be younger than its input files and still be up to date. `make` would not notice this because it just checks the file time stamps. In order to avoid this, a time stamp file for the `kimwitu` call is introduced. `Kimwitu` has to be called whenever this time stamp is outdated with respect to the `kimwitu` input files. This is formulated below.
The first lines of the text below introduce abbreviations for the `kimwitu` generated files and the auxiliary timestamp file for `kimwitu`.

```
KC_TIME = .kc_time_stamp
KC_OGEN = k.o csgiok.o unpk.o rk.o
KC_OSRC = ${KFILES:.k=.o}
KOBJS   = ${KC_OGEN} ${KC_OSRC}

# Kimwitu compilation (note: kc does not touch unchanged files)
${KC_TIME}: ${KFILES}
     ${KC} ${KFILES}
     date > ${KC_TIME}
```

It is straightforward to formulate the dependency (5), see below.

```
${EBNF}: %${EXE}: ${KC_TIME} ${LFILE:.l=.o} ${YFILE:.y=.o} ${KOBJS} %.o
     ${CC} ${CFLAGS} -o $@ ${LFILE:.l=.o} ${YFILE:.y=.o} ${KOBJS} $*.o
```

Now the dependencies are almost complete. However, we have to consider also dependencies between .c-Files and .h-Files. The most difficult one of those is that ebnf-lex.c includes y.tab.h which in turn is generated by yacc. Here the situation is the other way round then with kimwitu above: yacc will always generate y.tab.h, even when it is the same as before. To handle this case properly, we introduce an extra file ebnf-parse.h which is constructed from y.tab.h. Both files are compared and only in case of a difference ebnf-parse.h is updated. Instead of including y.tab.h we now use ebnf-parse.h for inclusion in ebnf-lex.c. Please note, that for constructing y.tab.h it suffices to construct ebnf-parse.c, because y.tab.h is generated also.

```
# the normal lex/yacc header trick
${YFILE:.y=.h} : y.tab.h; -cmp -s $@ $< || cp $< $@

y.tab.h : ${YFILE:.y=.c}
```

Finally, all dependencies between .c and .h files can be generated automatically using the C compiler. They are stored in a file named .depend.

```
depend: ${KC_TIME} ${LFILE:.l=.c} ${YFILE:.y=.c} ${YFILE:.y=.h}
     ${CC} -MM *.c > .depend
     @echo .depend is included
```

The file .depend is included into the makefile if it is existing, otherwise a warning message is generated.

```
DEPEND=${wildcard .depend}

ifneq "${DEPEND}" ""
include .depend
else
Makefile: MM
MM: ; @echo "**************** You must make depend first ******************"
endif
```

This concludes the makefile.

The EBNF main file

It remains to present the main file for the EBNF analyser, which calls all the elements introduced before.

```
1.  #include <stdio.h>
2.  #include "k.h"
3.  #include "rk.h"
4.  #include "unpk.h"

5.  extern int yyparse();
6.  extern void init_symtab();

7.  /* The syntax tree root */
8.  syntax TheSyntax;
9.  extern symtab TheSymtab;

10. void printer_f(const char *s, uview_enum v) { printf("%s", s); }
11. void dummy_printer_f(const char *s, uview_enum v) {}

12. int main()
13. { KC_Printer printer(printer_f), dummy_printer(dummy_printer_f);
14.   fprintf(stderr,"EBNF analysis\n");
15.   if (!yyparse())
16.   { init_symtab();
17.     TheSyntax->unparse( dummy_printer, create_symtab );
18.     TheSymtab->unparse( dummy_printer, check_symtab );
19.     TheSyntax = TheSyntax->rewrite( basic_rewrite );
20.     TheSyntax->unparse(printer, pretty);
21.     TheSymtab->unparse(printer, pretty);
22.     return 0;
23.   } else return 1;
24. } /* main */
```

Lines 1-9: Declare all the external parts that are generated by lex, yacc and kimwitu.
Lines 10-11: Introduce two printing functions: one really printing and one without output (see also lines 17-18).
Lines 15-21: Process the input according to the methodology.

2.3.5 ASM Workbench

For the implementation of the ASM parts of the semantics we use a tool called ASM workbench. This tool is provided by the university of Paderborn, see also [23]. There is a textual input format for this tool, which has to be generated by the appropriate back end tool that cares for the ASM part of the formal semantics definition. There are several differences between the ASM used by the formal semantics definition and the ASM used within the workbench. However, it is possible to use the workbench such that the wanted behaviour as used in the formal semantics definition can be achieved.

2.3.5.1 Input Format of the ASM Workbench

The input format of the ASM workbench is defined e.g. in [24]. We will not include it in detail here. In order to give you a feeling for the workbench input, we will show the translation of the RMS example (see Section 2.1.5).

```
// ASM text for workbench
freetype Agent == { generator_Agent : INT }
typealias Nat == INT
typealias Boolean == BOOL
external function Self: Agent
freetype TOKEN == { generator_Token: INT }
static function bound_Token == 1000
static function Token == { generator_Token(i) | i in { 1..bound_Token } }

freetype Mode == { exclusive, shared }

dynamic function mode: Agent -> Mode
initially MAP_TO_FUN emptymap

dynamic function owner: TOKEN -> Agent
initially MAP_TO_FUN emptymap

dynamic function Stop: Agent -> Boolean
initially MAP_TO_FUN emptymap

derived function Idle(a) ==
  ((mode(a) = undef) and (forall t in Token: (owner(t) != a)))

derived function Waiting(a) ==
  ((mode(a) != undef) and (forall t in Token: (owner(t) != a)))

derived function Busy(a) ==
  ((mode(a) != undef) and (exists t in Token: (owner(t) = a)))

derived function Available(t) == (owner(t) = undef)

transition Resource_Management_Program_SharedAccess ==
  if ((mode(Self) = shared) and Waiting(Self)) then
    choose t in Token with Available(t) owner(t):= Self
    endchoose
  endif

transition Resource_Management_Program_ExclusiveAccess ==
  if ((mode(Self) = exclusive) and (forall t in Token: Available(t))) then
    do forall t in Token owner(t):= Self
    enddo
  endif

transition Resource_Management_Program_ReleaseAccess ==
  if Stop(Self) then
    block
      mode(Self):= undef
      do forall t in Token with (owner(t) = Self) owner(t):= undef
      enddo
    endblock
  endif

(* main program *)
transition Resource_Management_Program ==
  block
    Resource_Management_Program_SharedAccess
    Resource_Management_Program_ExclusiveAccess
    Resource_Management_Program_ReleaseAccess
  endblock
```

2.3.5.2 Differences to ASM Definition

The ASM workbench has several features that impact on the ASM definition on the semantics.

Definition Order
The workbench follows a definition-before-use principle. This means, that every definition must be textually before any use of it. Moreover, each declaration must have a definition or initialisation attached to it.

Strong Typing
All workbench definitions are strongly typed. The type system for the workbench is based on standard ML and in some ways more restrictive than the typing used within the definitions in Part 4. In particular, it is not possible to use union domains in the workbench. Moreover, there is a distinction between (finite) sets and (infinite) domains which is not within the ASM definition.

Predefined Operators and Types
The predefined types of the workbench differ from those of the ASM definition in Section 2.1.6.

ASM Agents
The workbench implements only a single-agent ASM version. Multi-agent ASM could be emulated using interleaving. The current agent performing an action is selected by declaring *Self* to be an external function.

Shared Functions
There are only two kinds of dynamic functions within the current release of the workbench: external functions (which are **monitored** in our framework) and dynamic functions (**controlled** in our terminology). It is not possible to declare **shared** functions.

2.3.5.3 Running the Workbench

The workbench is based on and implemented in standard ML. It interacts with the outside via Tcl/Tk. After starting the workbench the program is loaded and ASM steps can be performed. If at any time an external function has to be evaluated, a so-called oracle is consulted that provides a value for the function. Currently, the oracle is implemented as a user interaction asking the user to provide the value. There is currently not much support for whole sequences of steps to be done automatically, but because the interworking is based on Tcl/Tk, this could be added using appropriate Tcl scripts.

Part 3: RSDL LANGUAGE DEFINITION

The name SDL stands for Specification and Description Language. SDL is an ITU[10] standardised language that is mainly used for telecommunication applications. The language SDL has a long standing tradition, the first standard appeared already in 1976. Since then, it was frequently adapted to emerging user needs. The current version is of 2000, and it is called SDL-2000. SDL has also a long tradition in formality, one could say its formality increased with every new version. Since 1988 there has always been a formal definition of the semantics of SDL. Due to the heavy changes from SDL-96 to SDL-2000 it did not seem feasible to change the old formal definition to cope also for the new requirements. So a new formal definition is being worked out. To this end, an initiative is under way to define such a new formal semantics (see also Section 6.5).
There are two representation forms for SDL, namely a graphical one and a textual one. These two representation forms are mostly equivalent apart from some parts of SDL that exist only in the textual part and some other parts that do only exist within the graphical part. In the context considered here, only the textual representation is used for the definition of the semantics.
This book will not explain SDL in full depth. It will rather concentrate on some aspects of the language that make it a complete specification language. All the other concepts of SDL can be found within the ITU standard Z.100 [12] and their semantics is explained within annex F of Z.100 [27]. Please note that the focus of this book is on the definition and the implementation of the semantics. Therefore, the concepts used in Part 5 of this book are complete with respect to the full language SDL. When more syntax and transformations for SDL are given, the technology will still work. However, in the scope of this book, a reduced SDL (RSDL) is considered. This RSDL is a proper subset of SDL, this means that any valid RSDL specification is also a valid SDL specification.
The following criteria have been used to select the RSDL subset of SDL:

1) It shall be possible to write simple meaningful examples within the scope of RSDL.
2) The informal language description should be at most 30 pages.
3) There should be transformations and static conditions in RSDL with complexity as in SDL.

The RSDL language description follows in style and contents closely the description in Z.100.

3.1 RSDL Short Description

In order to provide a quick overview of RSDL for SDL experts and to give a first impression of RSDL, this section summarises the features of RSDL. Moreover, an example shows how RSDL is used.

- The language RSDL describes a system using communicating state machines, called *blocks* (agents).
- The blocks communicate with each other using *signals* that are transferred over *channels*.
- Blocks may be defined using *block types*.
- For block types, connection points for the channels must be given,. They are called *gates*.
- Blocks may be *created* dynamically.
- The behaviour of a block (type) is given with a finite *state* machine.
- A state change is triggered by the reception of a signal (*input*) or by a condition (*continuous signal*).
- Timing conditions can be expressed using *timers*.
- A block may have local *variables*.
- A block may access local variables of another block using the concept of *remote variable*.
- RSDL provides no means to define data types. *Predefined data types* can be used, namely Boolean, Integer, Time, Duration and PId (block identities).

3.1.1 Daemon Game Example - Informal Description

The daemon game is often used as introductory example for SDL. We will use it here to show the expressiveness of RSDL. The following informal description of the game is taken from [14].

> The **Daemon Game** is a simple game having several players. The game is the system that is to be defined in RSDL. The players belong to the environment of the system.
> In the system there is a daemon that generates *Bump* signals randomly. A player has to guess whether the number of generated *Bump* signals is odd or even. The guess is made by sending a *Probe* signal to the system. The system replies by sending the signal *Win* if the number of generated *Bump* signals is odd, otherwise by the signal *Lose*.

[10] ITU is an abbreviation for International Telecommunication Union, which is the international telecommunication standardisation organisation.

The system keeps track of the score of each player. The score is initially 0. It is increased by 1 for each successful guess (signal *Win* is sent), and reduced by 1 for each unsuccessful guess (signal *Lose* is sent). A player can ask for the current value of the score by the signal Result, which is answered by the system with the signal *Score*.
Before a player can start playing, the player must log in. This is accomplished by the signal *Newgame*. A player logs out by the signal *Endgame*. The system allocates a player a unique identifier on logging in, and de-allocates it on logging out. The system cannot tell whether different identifiers are being used by the same player.

3.1.2 Daemon Game Example - Formal RSDL Description

In the RSDL formal description of the daemon game example the signals mentioned in the informal description are declared. The game is decomposed into two units: a monitor process and a game controller process. For each `Newgame` request, a game controller instance is created by the monitor. After reception of an `Endgame`, the corresponding game controller ceases to exists. The global structure is depicted below using the SDL graphical syntax. This figure is informal in the context of RSDL.

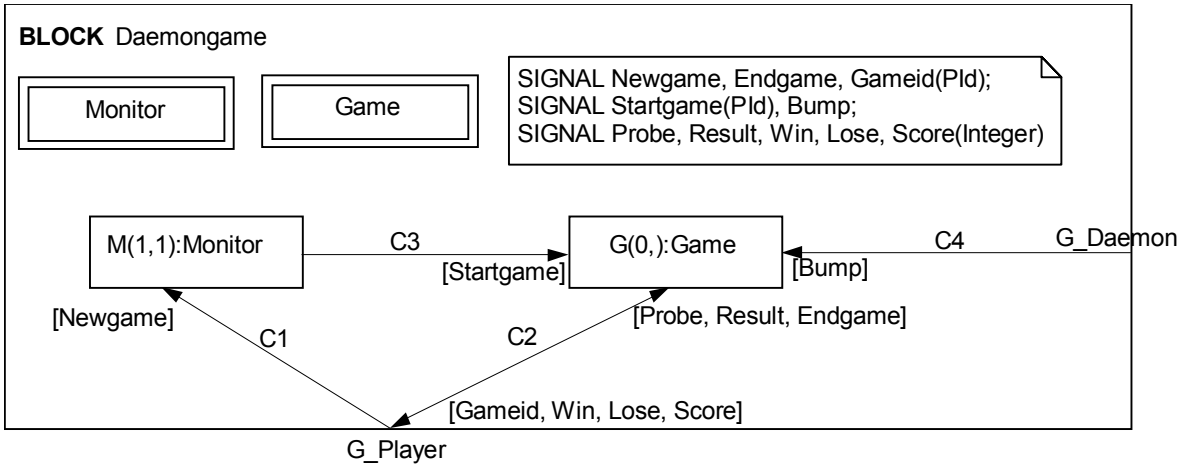


Figure 9: Overview of the RSDL Daemon Game Formalisation

The daemon game is connected to the environment via two gates, one for the communication with the players and one for the input from the daemon. All `Newgame` signals are routed to the Monitor using the channel `C1`. The channel `C2` is used to transport all the user signals to and from the game controller instances. The internal signal `Startgame` is sent from the monitor to the game controllers using the channel `C3`. Finally, the daemon `Bump` signals are transported on channel `C4`.
The daemon game contains exactly one monitor instance and as many game controller instances as there are active games. Initially, there are no active games. This structure is represented with the following RSDL block.

```
BLOCK Daemongame;

  SIGNAL Newgame, Endgame, Gameid(PId);
  SIGNAL Startgame(PId), Bump;
  SIGNAL Probe, Result, Win, Lose, Score(Integer);

  GATE G_Player IN WITH Newgame, Endgame, Probe, Result;
                OUT WITH Gameid, Win, Lose, Score;

  GATE G_Daemon IN WITH Bump;

  BLOCK TYPE Game REFERENCED;
  BLOCK TYPE Monitor REFERENCED;

  BLOCK M(1,1): Monitor;
  BLOCK G(0,): Game;

  CHANNEL C1
    FROM ENV VIA G_Player TO M VIA G_Newgame    WITH Newgame;
  ENDCHANNEL;

  CHANNEL C2
    FROM ENV VIA G_Player TO G VIA G_Playing    WITH Probe, Result, Endgame;
    FROM G VIA G_Playing TO ENV VIA G_Player    WITH Win, Lose, Score, Gameid;
  ENDCHANNEL;
```

```
  CHANNEL C3
    FROM M VIA G_Game TO G VIA G_Monitor
    WITH Startgame;
  ENDCHANNEL;

  CHANNEL C4
    FROM ENV VIA G_Daemon TO G VIA G_Bump
    WITH Bump;
  ENDCHANNEL;

ENDBLOCK Daemongame;
```

Now the block type monitor is specified in more detail. It has two gates, one for receiving `Newgame` indications and one for sending `Startgame` signals. Whenever the Monitor receives a `Newgame` signal, it creates a new game controller and sends a signal `Startgame` with the identity of the `Newgame` sender to the newly created instance. This behaviour has the following RSDL representation.

```
BLOCK TYPE Monitor;

  GATE G_Newgame IN WITH Newgame;
  GATE G_Game   OUT WITH Startgame;

  START;
    NEXTSTATE Idle;

  STATE Idle;
    INPUT Newgame;
      CREATE G;
      OUTPUT StartGame(SENDER) TO OFFSPRING;
      NEXTSTATE Idle;

ENDBLOCK TYPE Monitor;
```

The game controller has three interfaces (gates), one for receiving `Startgame` signals from the monitor, one for receiving `Bump` signals from the daemon and one for the communication with the player. Two local variables are declared, namely `score` to hold the current score and `MyPlayer` to hold the identity of the player.

```
BLOCK TYPE Game;

  GATE G_Monitor IN WITH Startgame;

  GATE G_Playing OUT WITH Gameid, Win, Lose, Score;
                 IN  WITH Probe, Result, Endgame;

  GATE G_Bump IN WITH Bump;

  DCL score Integer := 0;
  DCL MyPlayer PId;
```

After creation the game controller waits in an initial state for the `Startgame` signal. This signal contains the PId of the player. An acknowledge signal `Gameid` with the own PId is sent back to the player. Afterwards, the game starts in the losing state.

```
  START;
    NEXTSTATE Initstate;

  STATE Initstate;
    INPUT Startgame(MyPlayer);
      OUTPUT Gameid(SELF) TO MyPlayer;
      NEXTSTATE LoseState;
```

If the signal `Probe` is received in the losing state, the score is reduced by 1 and the player is informed with the signal `Lose`. If the player asks for the current score, a `Score` signal is sent. A similar action is taken in the winning state.

```
  STATE LoseState;
    INPUT Probe;
      TASK score:= score-1;
      OUTPUT Lose TO MyPlayer;
      NEXTSTATE LoseState;
    INPUT Result;
      OUTPUT Score(score) TO MyPlayer;
      NEXTSTATE WinState;
```

```
  STATE WinState;
    INPUT Probe;
      TASK score:= score+1;
      OUTPUT Win TO MyPlayer;
      NEXTSTATE WinState;
    INPUT Result;
      OUTPUT Score(score) TO MyPlayer;
      NEXTSTATE LoseState;
```

Switching between winning and losing is triggered by the `Bump` signal.

```
  STATE LoseState;
    INPUT Bump;
      NEXTSTATE WinState;

  STATE WinState;
    INPUT Bump;
      NEXTSTATE LoseState;
```

The `Endgame` signal causes the game to be finished.

```
  STATE WinState, LoseState;
    INPUT Endgame;
      STOP;

ENDBLOCK TYPE Game;
```

This concludes the daemon game specification in RSDL. The exact meaning of the RSDL constructs is given in the language description below.

3.2 Organisation of the Language Description

The RSDL language description is organised by topics described by an optional introduction followed by titled enumeration items for:

a) **Abstract grammar** – Described by abstract syntax and static conditions for well-formedness.
b) **Concrete grammar** – The concrete grammar used for RSDL. This grammar is described by the concrete syntax, static conditions and well-formedness rules for the concrete syntax, and the relationship of the concrete syntax with the abstract syntax.
c) **Semantics** – Gives meaning to a construct, provides the properties it has, the way in which it is interpreted and any dynamic conditions which have to be fulfilled for the construct to behave well in the RSDL sense.
d) **Model** – Gives the mapping for notations that do not have a direct abstract syntax and that are modelled in terms of other concrete syntax constructs. A notation that is modelled by other constructs is known as a shorthand, and is considered to be derived syntax for the transformed form.

If there is no text for a titled enumeration item, the whole item is omitted.
The remainder of this subclause describes the other special formalisms used in each titled enumeration item and the titles used. It can also be considered as an example of the typographical layout of first level titled enumeration items defined above where this text is part of an introductory section.

Abstract grammar
The abstract syntax notation is defined in Section 3.3.
If the titled enumeration item **Abstract grammar** is omitted, then there is no additional abstract syntax for the topic being introduced and the concrete syntax will map onto the abstract syntax defined by another numbered text section. The rules in the abstract syntax may be referred to from any of the titled enumeration items by use of the rule name in italics.
The rules in the formal notation may be followed by paragraphs that define conditions which must be satisfied by a well-formed RSDL definition and which can be checked without interpretation of an instance. The static conditions at this point refer only to the abstract syntax. Static conditions, that are only relevant for the concrete syntax are defined after the concrete syntax. Together with the abstract syntax, the static conditions for the abstract syntax define the abstract grammar of the language.

Concrete grammar
The concrete textual syntax is specified in the Backus-Naur Form of syntax description defined in Section 3.3.
The concrete syntax is followed by paragraphs defining the static conditions which must be satisfied in a well-formed text and which can be checked without interpretation of an instance. Static conditions (if any) for the abstract grammar also apply.
In many cases, there is a simple relationship between the concrete and abstract syntax, because the concrete syntax rule is simply represented by a single rule in the abstract syntax. When the same name is used in the abstract and concrete syntax in order to signify that they represent the same concept, then the text “<name> in

the concrete syntax represents *Name* in the abstract syntax" is implied in the language description and is often omitted. In this context, case is ignored but underlined semantic sub-categories (see Section 3.3) are significant.
Concrete syntax that is not a shorthand form is strict concrete syntax. The relationship from concrete syntax to abstract syntax is defined only for the strict concrete syntax.
The relationship between concrete syntax and abstract syntax is omitted if the topic being defined is a shorthand form that is modelled by other RSDL constructs (see **Model** below).

Semantics
Properties are relations between different concepts in RSDL. Properties are used in the well-formedness rules.
An example of a property is the set of visible identifiers of a block.
All instances have an identity property, but unless this is formed in some unusual way, this identity property is determined as defined by the general section on identities in Section 3.5. This is usually not mentioned as an identity property. Also, it has not been necessary to mention sub-components of definitions contained by the definition, since the ownership of such sub-components is obvious from the abstract syntax. For example, it is obvious that a block definition "has" enclosed blocks.
Properties are static if they can be determined without interpretation of an RSDL system specification, and are dynamic, if an interpretation of the same is required to determine the property.
The interpretation is described in an operational manner. Whenever there is a list in the Abstract Syntax, the list is interpreted in the order given. That is, it is described how the instances are created from the outermost block definition and how these instances are interpreted within an "abstract RSDL machine" . Lists are denoted in the Abstract Syntax by the suffixes "*" and "+" (see Section 3.3).
Dynamic conditions are conditions that must be satisfied during interpretation and cannot be checked without interpretation. Dynamic conditions may lead to errors. The further behaviour is undefined after the occurrence of an error.
Behaviour of the specification is produced by "interpreting" the RSDL. The word "interpret" is explicitly chosen (rather than an alternative such as "executed") to include both mental interpretation by a human and the interpretation of the RSDL by a computer.

Model
Some constructs are considered to be "derived concrete syntax" (or a shorthand notation) for other equivalent concrete syntax constructs. For example, omitting the name of a channel is derived concrete syntax for a channel with an implicit and unique name.
The properties of a shorthand notation are derived from the way it is modelled in terms of (or transformed to) the primitive concepts. In order to ensure easy and unambiguous use of the shorthand notations, and to reduce side effects when several shorthand notations are combined, these concepts are transformed in a specified order as defined in Section 3.13. The transformation order is also followed when defining the concepts in this section.

3.3 Grammar Notations

The following presentation forms are used to describe the syntax of RSDL.

3.3.1 Abstract Syntax

A definition in the abstract syntax can be regarded as a named composite object (a tree) defining a set of sub-components.
For example the abstract syntax for channel definition is

Channel-path :: *Originating-gate*
Destination-gate
Signal-identifier-**set**

which defines the domain for the composite object (tree) named *Channel-path.* This object consists of three sub-components, which in turn might be trees.
The abstract syntax definition

Agent-identifier = *Identifier*

expresses that an *Agent-identifier* is an *Identifier* and therefore cannot syntactically be distinguished from other identifiers.
An object might also be of some elementary (non-composite) domains. In the context of RSDL, these are:

a) Integer objects
Example
Number-of-instances :: *Initial-number* [*Maximum-number*]
Initial-number = *Int*
Maximum-number = *Int*
Number-of-instances denotes a composite domain containing one mandatory integer (*Int*) value and one optional integer ([*Int*]) denoting the initial number and the optional maximum number of instances.

b) Token objects

Token denotes the domain of tokens. This domain can be considered to consist of a potentially infinite set of distinct atomic objects for which no representation is required.

Example

Name :: *Token*

A name consists of an atomic object such that any *Name* can be distinguished from any other name.

The following concrete syntax operators (constructors) in BNF (see below) have the same use in the abstract syntax: "*" for possibly empty list, "+" for non-empty list, "|" for alternative, and "[" "]" for optional.

Parentheses are used for grouping of domains that are logically related.

Finally, the abstract syntax uses another postfix operator "-**set**" yielding a set (unordered collection of distinct objects).

Example

State-transition-graph :: *Start-node State-node*-**set** *Free-action*-**set**

A *State-transition-graph* consists of a *Start-node*, a set of *State-nodes* and a set of *Free-actions*.

3.3.2 Concrete Syntax

In the Backus-Naur Form for lexical rules the terminals are <space> and the ASCII printed characters. In the Backus-Naur Form for non-lexical rules, a terminal symbol is one of the lexical units defined in Section 3.4 (<name>, <special>, <composite special> or <keyword>). In non-lexical rules, a terminal can be represented by one of the following:

a) a keyword (such as **state**);
b) the character for the lexical unit, if it consists of a single character (such as "=");
c) the lexical unit name (such as <name>);
d) the name of a <composite special> lexical unit (such as <implies sign>).

To avoid confusion with the BNF grammar, the lexical unit names <asterisk> and <plus sign> are always used rather than the equivalent characters. Note that the special terminal <name> may also have semantics stressed as defined below.

The angle brackets and enclosed word(s) are either a non-terminal symbol or one of the lexical units. Syntactic categories are the non-terminals indicated by one or more words enclosed between angle brackets. For each non-terminal symbol, a production rule is given in concrete grammar. For example,

<block reference> ::=
    block <block name> **referenced** <end>

A production rule for a non-terminal symbol consists of the non-terminal symbol at the left-hand side of the symbol "::=" , and one or more constructs, consisting of non-terminal and/or terminal symbol(s) at the right-hand side. For example, <block reference> and <end> in the example above are non-terminals; **block**, <block name> and **referenced** are terminal symbols.

Sometimes the symbol includes an underlined part. This underlined part stresses a semantic aspect of that symbol. For example, <block name> is syntactically identical to <name>, but semantically it requires the name to be a block name.

At the right-hand side of the "::=" symbol several alternative productions for the non-terminal can be given, separated by vertical bars ("|"). For example,

<definition> ::=
    <agent definition> | <agent type definition>

expresses that a <definition> is an <agent definition> or an <agent type definition>.

Syntactic elements may be grouped together by using curly brackets ("{" and "}"), similar to the parentheses in the abstract syntax above. A curly bracketed group may contain one or more vertical bars, indicating alternative syntactic elements. For example,

<state machine graph> ::=
    <start> { <state> | <free action> }*

Repetition of syntactic elements or curly bracketed groups is indicated by an asterisk ("*") or plus sign ("+"). An asterisk indicates that the group is optional and can be further repeated any number of times; a plus sign indicates that the group must be present and can be further repeated any number of times. The example above expresses that <state machine graph> contains a <start> followed by any number of <state> or <free action>.

If syntactic elements are grouped using square brackets ("[" and "]"), then the group is optional. For example,

<identifier> ::= [<qualifier>] <name>

expresses that an <identifier> may, but need not, contain <qualifier>.

3.4 Lexical Rules

Lexical rules define lexical units. Lexical units are the terminal symbols of the **Concrete grammar**.

```
<lexical unit> ::=      <name>
                      | <note>
                      | <composite special>
                      | <special>
                      | <keyword>

<name> ::=              <underline>* <word> {<underline>+ <word>}* <underline>*
                      | {<decimal digit>}+ [ {<full stop>} <decimal digit>+ ]

<word> ::=              {<alphanumeric>}+

<alphanumeric> ::=      <letter>          |  <decimal digit>

<letter> ::=            <uppercase letter> | <lowercase letter>

<uppercase letter> ::=
          A | B | C | D | E | F | G | H | I | J | K | L | M
        | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<lowercase letter> ::=
          a | b | c | d | e | f | g | h | i | j | k | l | m
        | n | o | p | q | r | s | t | u | v | w | x | y | z

<decimal digit> ::=
          0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<note> ::=              <solidus> <asterisk> <note text> <asterisk>+ <solidus>

<note text> ::=       { <general text character>
                      | <other special>
                      | <asterisk>+ <not asterisk or solidus>
                      | <solidus>
                      | <apostrophe> }*

<not asterisk or solidus> ::= <general text character> | <other special> | <apostrophe>

<general text character> ::= <alphanumeric> | <other character> | <space>
<composite special> ::=   <greater than or equals sign>
                      | <implies sign>
                      | <is assigned sign>
                      | <less than or equals sign>
                      | <not equals sign>
                      | <qualifier begin sign>
                      | <qualifier end sign>

<greater than or equals sign> ::= <greater than sign> <equals sign>

<implies sign> ::=      <equals sign> <greater than sign>

<is assigned sign> ::=  <colon> <equals sign>

<less than or equals sign> ::= <less than sign> <equals sign>

<not equals sign> ::=   <solidus> <equals sign>

<qualifier begin sign> ::=  <less than sign> <less than sign>

<qualifier end sign> ::=    <greater than sign> <greater than sign>

<special> ::=           <solidus>          |  <asterisk>          |  <other special>

<other special> ::=     <left parenthesis> |  <right parenthesis>
                      | <plus sign>       |  <comma>           |  <hyphen>
                      | <colon>          |  <semicolon>
                      | <less than sign>  |  <equals sign>     |  <greater than sign>
```

<other character> ::= <exclamation mark> | <number sign> | <full stop>
| <quotation mark> | <dollar sign> | <percent sign>
| <ampersand> | <question mark> | <commercial at>
| <reverse solidus> | <circumflex accent> | <underline>
| <grave accent> | <vertical line> | <tilde>
| <left square bracket> | <right square bracket>
| <left curly bracket> | <right curly bracket>

<exclamation mark>	::= **!**	<quotation mark>	::= **"**
<left parenthesis>	::= **(**	<right parenthesis>	::= **)**
<asterisk>	::= *****	<plus sign>	::= **+**
<comma>	::= **,**	<hyphen>	::= **-**
<full stop>	::= **.**	<solidus>	::= /
<colon>	::= **:**	<semicolon>	::= **;**
<less than sign>	::= **<**	<equals sign>	::= **=**
<greater than sign>	::= **>**	<left square bracket>	::= **[**
<right square bracket>	::= **]**	<left curly bracket>	::= **{**
<right curly bracket>	::= **}**	<number sign>	::= **#**
<dollar sign>	::= **$**	<percent sign>	::= **%**
<ampersand>	::= **&**	<apostrophe>	::= **'**
<question mark>	::= **?**	<commercial at>	::= **@**
<reverse solidus>	::= \	<circumflex accent>	::= ^
<underline>	::= **_**	<grave accent>	::= `
<vertical line>	::= **|**	<tilde>	::= ~

<keyword> ::=

active	**and**	**block**
channel	**connect**	**connection**
create	**dcl**	**decision**
else	**endblock**	**endchannel**
endconnection	**enddecision**	**endstate**
env	**export**	**exported**
from	**gate**	**import**
in	**input**	**join**
mod	**nextstate**	**not**
now	**offspring**	**or**
out	**output**	**parent**
provided	**referenced**	**remote**
reset	**save**	**self**
sender	**set**	**signal**
start	**state**	**stop**
task	**timer**	**to**
type	**via**	**with**
xor		

<space> ::=

The characters in <lexical unit>s and in <note>s as well as the character <space> and control characters are defined by the International Reference Version of the International Reference Alphabet (Recommendation T.50), which is basically the same as ASCII. The lexical unit <space> represents the T.50 SPACE character (acronym SP), which (for obvious reasons) cannot be shown.
When an <underline> character is followed by one or more <space>s or control characters, all of these characters (including the <underline>) are ignored, e.g. A_ B denotes the same <name> as AB. This use of <underline> allows <lexical unit>s to be split over more than one line. This rule is applied before any other lexical rule.
A (non-space) control character may appear where a <space> may appear, and has the same meaning as a <space>.
Any number of <space>s may be inserted before or after any <lexical unit>. Inserted <spaces> or <note>s have no syntactic relevance, but sometimes a <space> or <note> is needed to separate one <lexical unit> from another.
In all <lexical unit>s uppercase <letter>s and lowercase <letter>s are distinct. Therefore AB, aB, Ab and ab represent four different <word>s. A <keyword> with all uppercase letters has the same use as the (lowercase) <keyword> with the same spelling (ignoring case), but a mixed case letter sequence with the same spelling as a <keyword> represents a <word>.

For conciseness within the grammar, a <keyword> as a terminal denotes the uppercase and the lowercase variant with the same spelling. For example, the concrete syntax terminator

endblock

represents the lexical alternatives

{ **endblock** | **ENDBLOCK** }

However, both alternatives are not considered to be distinct within the concrete grammar.
A <lexical unit> is terminated by the first character which cannot be part of <lexical unit> according to the syntax specified above. If a <lexical unit> can be both a <name> and a <keyword>, then it is a <keyword>.

For similarity with the SDL grammar the following production is introduced for RSDL.

<end> ::= <semicolon>

3.5 Visibility Rules, Names and Identifiers

Abstract grammar

Identifier	::	*Qualifier Name*
Qualifier	=	*Path-item**
Path-item	=	*Agent-type-qualifier*
	|	*Agent-qualifier*
Agent-type-qualifier	::	*Agent-type-name*
Agent-qualifier	::	*Agent-name*
Agent-type-name	=	*Name*
Agent-name	=	*Name*
Name	::	*Token*

Concrete grammar

<identifier> ::= [<qualifier>] <name>

<qualifier> ::= <qualifier begin sign> <path item> { / <path item> }* <qualifier end sign>

<path item> ::= <scope unit kind> <name>

<scope unit kind> ::= **block** | **block type**

Scope units are defined by the following non-terminal symbols of the concrete grammar.

<agent definition>
<agent type definition>

A scope unit has a list of definitions attached. Each of the definitions defines one or more entities belonging to a certain entity kind and having an associated name, including <textual gate definition>s, <agent formal parameters>s, and <formal variable parameters> contained in the scope unit.
Entities can be grouped into entity kinds. The following entity kinds exist:

a) agents (blocks);
b) agent types (block types);
c) channels, gates;
d) signals, timers;
e) variables (including formal parameters), literals, data types;
f) remote variables;

A <reference definition> is an entity after the transformation step for <referenced definition> (see Section 3.13).
Each entity is said to have its defining context in the scope unit which defines it.
Entities are referenced by means of <identifier>s. The <qualifier> within an <identifier> specifies uniquely the defining context of the entity.
The <qualifier> reflects the hierarchical structure from the outermost block level to the defining context, such that the outermost block level is the leftmost textual part. The *Identifier* of an entity is then represented by the qualifier, and the name of the entity. All entities of the same kind must have different *Identifiers*. Consequently, no two definitions in the same scope unit and belonging to the same entity kind can have the same <name>.
<operation name>s, <state name>s, <connector name>s, and <gate name>s occurring in channel definitions have special visibility rules and cannot be qualified. Other special visibility rules are explained in the appropriate sections.
An entity can be referenced by using an <identifier>, if the entity is visible. An entity is visible in a scope unit if:

a) it has its defining context in that scope unit; or
b) the entity is visible in the scope unit which defines that scope unit.

It is allowed to omit some of the leftmost <path item>s, or the whole <qualifier> of an <identifier> if the omitted <path item>s can be uniquely expanded to a full <qualifier>.
When the <name> part of an <identifier> denotes an entity, the <name> is bound to an entity that has its defining context in the nearest enclosing scope unit in which the <qualifier> of the <identifier> is the same as the rightmost part of the full <qualifier> denoting this scope unit (resolution by container). If the <identifier> does not contain a <qualifier>, then the requirement on matching of <qualifier>s does not apply.
The binding of a <name> to a definition through resolution by container proceeds in the following steps, starting in the scope unit where the <identifier> appears:

a) if a unique entity exists in the current scope unit with the same <name> and entity kind and matching <qualifier>s, the <name> is bound to that entity; otherwise
b) resolution by container is attempted in the scope unit which defines the current scope unit.

In the Concrete grammar, the optional name in a definition after the ending keywords (**endblock**, etc.) in definitions must be syntactically the same as the name following the corresponding commencing keyword (**block**, etc., respectively).

3.6 General Structure

3.6.1 Framework

An <rsdl specification> can be described as a monolithic <system specification> and <referenced definition>s. A <referenced definition> is a definition that has been removed from its defining context to gain overview within one system description. It is "inserted" into exactly one place (the defining context) using a reference.

Abstract grammar

RSDL-specification :: *Agent-type-definition Agent-definition*

Concrete grammar

<rsdl specification> ::= <system specification> <referenced definition>*

<system specification> ::= <textual system specification>

<textual system specification> ::=
<agent definition>
| <agent type definition> <textual typebased agent definition>

Semantics
An *RSDL-specification* has the semantics of the *Agent-definition*.

3.6.2 Referenced Definition and References

Concrete grammar

<referenced definition> ::= <definition>

<definition> ::= <agent definition>
| <agent type definition>

<agent type reference> ::= <block type reference>

<block type reference> ::= **block type** <block type name> **referenced** <end>

<agent reference> ::= <block reference>

<block reference> ::= **block** <block name> **referenced** <end>

For each <referenced definition> there must be a reference in the associated <rsdl specification>.
A <name> is present in a <referenced definition> after the initial keyword(s). For each reference there must exist exactly one <referenced definition> with the same <name> and entity kind as the reference.

Model
Before the properties of an <rsdl specification> are derived, each reference is replaced by the corresponding <referenced definition>.

3.7 Agents and Agent Types

This clause introduces a number of language mechanisms to support the modelling of application specific phenomena by instances and application specific concepts by types.
The language mechanisms introduced provide:
a) (pure) type definitions that may be defined anywhere in a system; and
b) typebased instance definitions that define instances or instance sets according to types.

3.7.1 Block Types

There is a distinction between definition of instances (or sets of instances) and definition of types in RSDL descriptions. This clause introduces type definitions for agents, and corresponding instance specifications. An agent type definition is not connected (by channels) to any instances; instead, agent type definitions introduce gates (Section 3.8). These are connection points on the typebased instances for channels.
A type defines a set of properties, which are used by instances of the type.
An instance (or instance set) always has a type, which is implied if the instance is not explicitly based on a type, i.e. a block definition has an implied equivalent anonymous block type.

Abstract grammar

Agent-type-definition :: *Agent-type-name*
*Signal-definition-**set***
*Timer-definition-**set***
*Variable-definition-**set***
*Agent-type-definition-**set***
*Agent-definition-**set***
*Gate-definition-**set***
*Channel-definition-**set***
[*State-transition-graph*]

State-transition-graph :: *Start-node*
*State-node-**set***
*Free-action-**set***

Agent-type-identifier = *Identifier*

If *Agent-definition*-**set** is not empty, *Variable-definition*-**set** must be empty and *State-transition-graph* must not be present. If *State-transition-graph* is present, *Agent-definition*-**set** and *Channel-definition*-**set** must be empty.

Concrete grammar

<agent type definition> ::= <block type definition>

<agent type structure> ::= { <entity in agent>
| <channel definition>
| <channel to channel connection>
| <gate in definition>
| <agent definition>
| <agent reference>
| <textual typebased agent definition> }*
[<agent type body>]

<agent type body> ::= <state machine graph>

<state machine graph>::= <start> { <state> | <free action> } *

<entity in agent> ::= <signal definition>
| <variable definition>
| <remote variable definition>
| <timer definition>
| <agent type definition>
| <agent type reference>

<block type definition> ::= <block type heading> <end> <agent type structure>
endblock type [<block type name>] <end>

<block type heading> ::= **block type** <block type name>

Semantics
An *Agent-type-definition* defines an agent type. All agents of an agent type have the same properties as defined for that agent type.

Signals mentioned in <output>s of the state machine of an agent type must be in the <signal list> of a gate in the direction from the agent type.
The properties defined in an *Agent-type-definition* such as the *Agent-definition-**set***, and *Gate-definition-**set*** determine the properties of any *Agent-definition* based on the type.
A <block type definition> defines a block type.

3.7.2 Typebased Block Definition

Abstract grammar

Agent-definition	::	*Agent-name*
		Number-of-instances
		Agent-type-identifier
Number-of-instances	::	*Initial-number* [*Maximum-number*]
Initial-number	=	*Int*
Maximum-number	=	*Int*
Agent-identifier	=	*Identifier*

Concrete grammar

<textual typebased agent definition> ::=
<textual typebased block definition>

<textual typebased block definition> ::=
block <typebased block heading> <end>

<typebased block heading> ::=
<block name> [<number of instances>] <colon> <block type expression>

<type expression> ::= <base type>

<base type> ::= <identifier>

<number of instances> ::= **(** [<initial number>] [**,** [<maximum number>]] **)**

<initial number> ::= <Integer name>

<maximum number> ::= <Integer name>
The initial number of instances and maximum number of instances contained in *Number-of-instances* are derived from <number of instances>. If <initial number> is omitted, then <initial number> is 1. If <maximum number> is omitted, then <maximum number> is unbounded.
The <initial number> of instances must be less than or equal to <maximum number> and <maximum number> must be greater than zero.

Semantics
A typebased block definition defines an *Agent-definition* based on a block type.
Blocks are always defined based on types. Types, on the other hand, may be arbitrarily nested in scopes.
An agent definition defines an (arbitrarily large) set of agents (blocks). An agent is characterised by having variables, a state machine or sets of contained agents.
A *system* is the outermost *block*.

3.7.3 Direct Agent Definitions

A block is defined by a <block definition>.
The instances contained within a block instance are interpreted concurrently and asynchronously with each other. All communication between different contained instances within a block is performed asynchronously using signal exchange, either explicitly or implicitly.

Concrete grammar

<agent definition> ::= <block definition>

<agent structure> ::= <agent type structure>

<block definition> ::= <block heading> <end> <agent structure>
endblock [<block name>] <end>

<block heading> ::= **block** <block name> <agent instantiation>

<agent instantiation> ::= [<number of instances>]

Semantics

An *Agent-definition* has a name which can be used in qualifiers in conjunction with **block**.

An agent definition defines a set of agents. Several instances of the same agent set may exist at the same time and be interpreted asynchronously and in parallel with each other and with instances of other agent sets in the system.

The first value in the *Number-of-instances* represents the number of instances of the agent set which exist when the system or containing entity is created (initial instances), the second value represents the maximum number of simultaneous instances of the agent set.

An agent instance may have a communicating extended finite state machine defined by its state machine definition. Whenever the state machine is in a state, on input of a given signal it will perform a certain sequence of actions, denoted as a transition. The completion of the transition results in the state machine of the agent instance waiting in another state, which is not necessarily different from the first one.

When an agent is interpreted, the initial agents it contains are created. The signal communication between the finite state machines of these initial agents, the finite state machine of the agent and their environment commences only when all the initial agents have been created. The time taken to create an agent may or may not be significant.

Agent instances exist from the time that the containing agent is created or they can be created by create request actions of agents being interpreted; their interpretations start when their start action is interpreted; they may cease to exist by performing stop actions.

When the state machine of an agent interprets a stop, the agent ceases to exist.

Signals received by agent instances are denoted as input signals, and signals sent from agent instances are denoted as output signals.

Accessing remote variables also corresponds to exchange of signals, see Section 3.8.5.

Signals may be consumed by the state machine of an agent instance only when it is in a state.

Exactly one input port is associated with the finite state machine of each agent instance. Signals sent to a container agent will be delivered to a contained agent according to the channel structure.

The finite state machine of an agent is either waiting in a state or active, performing a transition. For each state, there is a save signal set. When waiting in a state, the first input signal whose identifier is not in the save signal set is taken from the queue and consumed by the agent.

The input port may retain any number of input signals, so that several input signals can be queued for the finite state machine of the agent instance. The set of retained signals are ordered in the queue according to their arrival time. If two or more signals arrive on different paths "simultaneously", they are arbitrarily ordered.

When the agent is created, its finite state machine is given an empty input port, and local variables of the agent are created.

When a container agent instance is created, the initial agents of the contained agent sets are created. If the container is created by a <create request>, **parent** of the contained agents (see **Model** below) receives the pid of the container.

A block definition is an agent definition that defines containers for one or more block definitions.

A block instance is an instantiation of a block type defined by an *Agent-definition*. To interpret a block instance is to:

a) interpret the contained agents and their connected channels, or
b) interpret the state machine of the block (if present).

In a block with a finite state machine, the finite state machine is interpreted.

Variables of a block cannot be accessed from other blocks.

Model

An *Agent-definition* has an implied anonymous agent type that defines the properties of the agent.

In all agent instances, four anonymous variables of the pid sort are declared and are, in the following, referred to by **self**, **parent**, **offspring** and **sender**. They give a result for:

a) the agent instance (**self**);
b) the creating agent instance (**parent**);
c) the most recent agent instance created by the agent instance (**offspring**);
d) the agent instance from which the last input signal has been consumed (**sender**).

These anonymous variables are accessed using pid expressions as further explained in Section 3.12.4.

For all initial agent instances **parent** is initialised to the container agent. For the system agent, **parent** is initialised to null.

For all newly created agent instances, **sender** and **offspring** are initialised to null.

3.8 Communication

3.8.1 Gate

Gates are defined in agent types (block types) and represent connection points for channels, connecting instances of these types (as defined in Section 3.7.2) with other instances.
It is possible also to define gates in agents and this represents a notation for specifying that the considered entity has a named connection point.

Abstract grammar

Gate-definition	::	*Gate-name* *In-signal-identifier**-set*** *Out-signal-identifier**-set***
Gate-name	=	*Name*
In-signal-identifier	=	*Signal-identifier*
Out-signal-identifier	=	*Signal-identifier*

Concrete grammar

<gate in definition> ::= <textual gate definition>

<textual gate definition> ::= **gate** <gate> <gate constraint> <end> [<gate constraint> <end>]

<gate> ::= <gate name>

<gate constraint> ::= { **out** | **in** } **with** <signal list>
out or **in** denotes the direction of <signal list>, from or to the type respectively.
An *In-signal-identifier* represent an element in the <signal list> to the gate. An *Out-signal-identifier* represents an element in the <signal list> from the gate.
A channel connected to a gate must be compatible with the gate constraint. A channel is compatible with a gate constraint if the set of signals on the channel is equal to or is a subset of the set of signals specified for the gate in the respective direction.
Where two <gate constraint>s are specified one must be in the reverse direction to the other.

Semantics
Gates in type definitions are connection points for channels.

3.8.2 Channel

Abstract grammar

Channel-definition	::	*Channel-name* *Channel-path**-set***
Channel-path	::	*Originating-gate* *Destination-gate* *Signal-identifier**-set***
Originating-gate	=	*Gate-identifier*
Destination-gate	=	*Gate-identifier*
Gate-identifier	=	*Identifier*
Channel-name	=	*Name*

The *Channel-path***-set** contains at least one *Channel-path* and no more than two. When there are two paths the channel is bi-directional and the *Originating-gate* of each *Channel-path* must be the same as the *Destination-gate* of the other *Channel-path*.
If the *Originating-gate* and the *Destination-gate* are in the same agent, the channel must be unidirectional (there must be only one element in the *Channel-path**-set***).
The *Originating-gate* or *Destination-gate* must be defined in the same scope unit in the abstract syntax in which the channel is defined.
A channel is allowed to connect the two directions of a bi-directional gate to each other.

Concrete grammar

<channel definition> ::= **channel** [<channel name>] <channel path> [<channel path>]
endchannel [<channel name>] <end>

<channel path> ::= **from** <channel endpoint>
to <channel endpoint> **with** <signal list> <end>

<channel endpoint> ::= { <agent identifier> | **env** } [<via gate>]

<via gate> ::= **via** <gate>

<signal list> ::= <signal list item> { **,** <signal list item>}*

<signal list item> ::= <signal identifier> | <timer identifier> | <remote variable identifier>

The ending <channel name> may only be specified if the starting <channel name> is specified. If the starting <channel name> is not specified, the channel cannot be referred to by name.

<gate> must be specified if:

a) <channel endpoint> denotes a connection to a <textual typebased agent definition> in which case the <gate> must be defined directly in the agent type for that agent; or
b) **env** is specified and the channel is defined in an agent type in which case the <gate> must be defined in this agent type respectively.

If <gate> is specified, the channel is connected to that gate. The gate and the channel must have at least one common element in their signal lists in the same direction. If no <gate> is specified, the following rule applies:

> If the channel endpoint is an agent, then that agent must contain a <channel to channel connection> for the channel and the channel is connected to the implicit gate introduced by the <channel to channel connection>.

The <signal list> which is constructed by replacing all the <remote variable identifier>s by one of the implicit signals each of them denotes (Section 3.8.5), corresponds to a *Signal-identifier*-**set** in the **Abstract grammar**.

A <signal list item> which is an <identifier> denotes a <signal identifier> or <timer identifier> if this is possible according to the visibility rules or else a <remote variable identifier>.

Semantics

A *Channel-definition* represents a transportation path for signals (including the implicit signals implied by remote variables, see Section 3.8.5). A channel can be considered as one or two independent unidirectional channel paths between two agents or between an agent and its environment.

The *Signal-identifier*-**set** in each *Channel-path* in the *Channel-definition* contains the signals that may be conveyed on that *Channel-path*.

Signals conveyed by channels are delivered to the destination endpoint.

Signals are presented at the destination endpoint of a channel in the same order they have been presented at its origin. If two or more signals are presented simultaneously to the channel, they are arbitrarily ordered.

A channel may delay the signals conveyed by the channel. That means that a First-In-First-Out (FIFO) delaying queue is associated with each direction in a channel. When a signal is presented to the channel, it is put into the delaying queue. After an indeterminate and non-constant time interval, the first signal instance in the queue is released and given to one of the endpoints which is connected to the channel.

Several channels may exist between the same two endpoints. The same signal type can be conveyed on different channels.

A remote variable on a channel is mentioned as outgoing from an importer and incoming to an exporter.

Model

If the <channel name> is omitted from a <channel definition>, the channel is implicitly and uniquely named.

3.8.3 Connection

Concrete grammar

<channel to channel connection> ::=
connect <external channel identifiers>
and <channel identifiers> <end>

<external channel identifiers> ::=
<channel identifier> { **,** <channel identifier>}*

<channel identifiers> ::=
<channel identifier> { **,** <channel identifier>}*

No channel may be mentioned after the keyword **and** in more than one <channel to channel connection> of a given scope unit.

No channel may be mentioned before the keyword **and** in more than one <channel to channel connection> of a given scope unit.

Semantics
The <channel identifier>s in an <external channel identifiers> part of a <channel to channel connection> must denote channels connected to the enclosing agent. Each channel connected to the enclosing agent must be mentioned in the <external channel identifiers> part of one <channel to channel connection>.
Each channel identified by a <channel identifier> in a <channel identifiers> part of a <channel to channel connection> must be defined in the same agent in which the <channel to channel connection> is defined and it must have the boundary of that agent as one of its endpoints. Each channel defined in the surrounding agent and having the agent as one of its endpoints, must be mentioned in the <channel identifiers> part of exactly one <channel to channel connection>.

Model
Connections are shorthand constructs and are transformed to gates.
Each different <channel to channel connection> in a given scope unit defines one implicit gate on the scope unit. All channels in the <channel to channel connection> are connected to that gate in their respective scope units. For channels to the environment of the system agent also an implicit gate is defined. The gate constraints of the implicit gate are derived from the channels connected to the gate.
The name of the gate is a unique and unambiguous derived name. In the surrounding scope unit the <channel definition> that is identified by the <channel identifier> is extended with a <via gate> part. The <via gate> part is added to the <channel endpoint> that references the current scope unit and it mentions the implicit gate. Inside the scope unit the channels that are associated with the external channel by means of the <channel to channel connection> are modified, by extending the <channel endpoint> that mentions **env** with a <via gate> part for the implicit gate.

3.8.4 Signal

Abstract grammar

Signal-definition	::	*Signal-name* *Sort-name**
Signal-identifier	=	*Identifier*
Signal-name	=	*Name*

Concrete grammar

<signal definition>::= **signal** <signal definition item> { **,** <signal definition item> }* <end>

<signal definition item> ::= <signal name> [<sort list>]

<sort list> ::= **(** <sort> { **,** <sort>}* **)**

Semantics
A signal instance is a flow of information between agents, and is an instantiation of a signal type defined by a signal definition. A signal instance can be sent by either the environment or an agent and is always directed to either an agent or the environment. A signal instance is created when an *Output-node* is interpreted and ceases to exist when an *Input-node* is interpreted.

3.8.5 Remote variables

In RSDL, a variable is always owned by, and local to, an agent instance. Normally the variable is visible only to the agent instance which owns it. If an agent instance in another agent needs to access the data items associated with a variable, a signal interchange with the agent instance owning the variable is needed.
This can be achieved by the following shorthand notation, called imported and exported variables.

Concrete grammar

<remote variable definition> ::=
remote <remote variable name> {**,**<remote variable name>}* <sort>
{**,** <remote variable name> {**,** <remote variable name>}* <sort>}*
<end>

<import> ::= <variable> <is assigned sign>
import (<remote variable identifier> <communication constraints> **)**

<export> ::= **export (** <variable identifier> { **,** <variable identifier> }* **)**

A remote variable mentioned in an <import> must be in the complete output set (see Section 3.13) of an enclosing agent type or agent set.
The <variable identifier> in <export> must denote a variable defined with **exported**.

Semantics

A <remote variable definition> introduces the name and sort for imported and exported variables.

An exported variable definition is a variable definition with the keyword **exported**.

The association between an imported variable and an exported variable is established by both referring to the same <remote variable definition>.

Imported variables are specified as part of the output signals of the enclosing agent. Exported variables are specified as part of the input signals of the enclosing agent.

The agent instance which owns a variable whose data items are exported to other agent instances is called the exporter of the variable. Other agent instances which use these data items are known as importers of the variable. The variable is called exported variable.

An agent instance may be both importer and exporter of the same remote variable.

a) *Export operation*

Exported variables have the keyword **exported** in their <variable definition>s, and have an implicit copy to be used in import operations.

An export operation is the interpretation of an <export> by which an exporter discloses the current result of an exported variable. An export operation causes the storing of the current result of the exported variable into its implicit copy.

b) *Import operation*

An import operation is the interpretation of an <import> by which an importer accesses the result of an exported variable. The result is stored in the result variable denoted by the <variable> in the <import>. The exporter containing the exported variable is specified by the <destination> in the <import>. If no <destination> is specified then the import is from an arbitrary agent instance exporting the same remote variable. The association between the exported variable in the exporter and the implicit variable in the importer is specified by referring to the same remote variable in the export variable definition and in the <import>.

Model

An import operation is modelled by exchange of implicitly defined signals. The importer sends a signal to the exporter, and waits for the reply. In response to this signal the exporter sends a signal back to the importer with the result contained in the implicit copy of the exported variable.

If a default initialisation is attached to the export variable, then the implicit copy is also initialised with the same result as the export variable.

There are two implicit <signal definition>s for each <remote variable definition> in a block definition. The <signal name>s in these <signal definition>s are denoted by xQUERY and xREPLY, where *x* denotes the <name> of the <remote variable definition>. The signals are defined in the same scope unit as the <remote variable definition>. The signal xQUERY has no arguments and xREPLY has one arguments of the sort of the variable. The implicit copy of the exported variable is denoted by imcx.

On each channel mentioning the remote variable, the remote variable is replaced by xQUERY. For each such channel, a new channel is added in the opposite direction; this channel carries the signal xREPLY.

a) *Importer*

The <import>

v := **import** (x **to** destination)

is transformed to the following, where the **to** clause is omitted if the destination is not given:

output xQUERY **to** destination;
wait in state xWAIT, saving all other signals;
input xREPLY(v);

b) *Exporter*

To all <state>s of the exporter, excluding implicit states derived from import, the following <input part> is added:

input xQUERY;
output xREPLY(imcx) **to sender**;
nextstate *the state containing this input*;

The <export>

export x

is transformed to the following:

task imcx := x;

3.9 Behaviour

3.9.1 States

Abstract grammar

Start-node	::	*Transition*
State-node	::	*State-name* *Save-signalset* *Input-node-**set*** *Continuous-signal-**set***
State-name	=	*Name*

*State-node*s within a *State-transition-graph* must have different *State-name*s.
The *Signal-identifier*s in the *Input-node*-**set** must be distinct.

Concrete grammar

<start> ::= **start** <end> <transition>

<state> ::= <basic state>

<basic state> ::= **state** <state list> <end>
{ <input part>
| <save part>
| <continuous signal> }*
[**endstate** [<state name>] <end>]

<state list> ::= <state name> { **,** <state name> }*

When the <state list> contains one <state name> then the <state name> represents a *State-node*. For each *State-node*, the *Save-signalset* is represented by the <save part>. For each *State-node*, the *Input-node*-**set** is represented by the <input part> and any implicit input signals.
A <state name> may appear in more than one <state> of a body.
The optional <state name> ending a <state> may be specified only if the <state list> in the <state> consists of a single <state name> in which case it must be that <state name>.

Semantics
The *Transition* of the *Start-node* is interpreted when the enclosing agent starts to exist.
A state represents a particular condition in which the state machine of an agent may consume a signal instance. If a signal instance is consumed, the associated transition is interpreted. A transition may also be interpreted as the result of a continuous signal.
A signal is enabled when it is not in the *Save-signalset* of the current state. A *Continuous-signal* is enabled when it yields the value *True*. Signals that are not mentioned in the *Save-signalset* or in any *Input-node* are implicitly discarded.
For each state, the *Save-signal*s, *Input-node*s, and *Continuous-signal*s are interpreted in the following order:

a) in the order of the signals on the input port:
   1) if the current signal is enabled, this signal is consumed; otherwise,
   2) the next signal on the input port is selected.
b) if no enabled signal was found, for the *Continuous-signal*s in any order:
   1) the *Continuous-expression* contained in the current *Continuous-signal* is interpreted;
   2) if the current continuous signal is enabled, this signal is consumed; otherwise,
   3) the next continuous signal is selected.
c) if no enabled signal was found, the state machine waits in the state until another signal instance is received. If the state has continuous signals, these steps are repeated even if no signal is received.

Model
When the <state list> of a <state> contains more than one <state name>, a copy of that <state> is created for each such <state name>. Then the <state> is replaced by these copies.
When several <state>s contain the same <state name>, these <state>s are concatenated into one <state> having that <state name>.

3.9.2 Trigger Events

A continuous signal interprets a Boolean expression and the associated transition is interpreted when the expression returns the predefined Boolean value true.
A save specifies a set of signal identifiers whose instances are not relevant to the agent in the state to which the save is attached, and which need to be saved for future processing.

Abstract grammar

Input-node	::	*Signal-identifier* [*Variable-identifier*]* *Transition*
Variable-identifier	=	*Identifier*
Continuous-signal	::	*Continuous-expression Transition*
Continuous-expression	=	*Boolean-expression*
Boolean-expression	=	*Expression*
Save-signalset	=	*Signal-identifier**-set***

The length of the list of optional *Variable-identifier*s must be the same as the number of *Sort-names* in the *Signal-definition* denoted by the *Signal-identifier*. The sorts of the variables must correspond by position to the sorts of the data items that can be carried by the signal.

Concrete grammar

<input part> ::= **input** <input list> <end>
<transition>

<input list> ::= <stimulus> { **,** <stimulus> }*

<stimulus> ::= <signal list item> [**(** [<variable>] { **,** [<variable>] }* **)**]

<continuous signal> ::= **provided** <continuous expression> <end> <transition>

<continuous expression> ::=<Boolean expression>

<save part> ::= **save** <save list> <end>

<save list> ::= <signal list>

A <signal list item> in a <stimulus> must not denote a <remote variable identifier>.
When the <input list> contains one <stimulus>, then the <input part> represents an *Input-node*. In the **Abstract grammar**, timer signals (<timer identifier>) are also represented by *Signal-identifier*. Timer signals and ordinary signals are distinguished only where appropriate, as in many respects they have similar properties. The exact properties of timer signals are defined in Section 3.10.6. A <save list> represents the *Signal-identifier*-**set**.

Semantics
An input allows the consumption of the specified input signal instance. The consumption of the input signal makes the information conveyed by the signal available to the agent. The variables associated with the input are assigned the data items conveyed by the consumed signal.
The data items are assigned to the variables from left to right. If there is no variable associated with the input for a sort specified in the signal, the corresponding data item is discarded. If there is no data item associated with a sort specified in the signal, the corresponding variable becomes "undefined".
The sender of the consuming agent (see Section 3.7.3 **Model**) is given the pid of the originating agent, as carried by the signal instance. Signal instances flowing from the environment to an agent instance within the system will always carry a pid different from any in the system.
The *Continuous-expression* is interpreted upon entering the state to which its *Continuous-signal* is associated, and while waiting in the state, whenever no <stimulus> of an attached <input list> is found in the input port. If the *Continuous-expression* returns the predefined Boolean value true, the continuous signal is enabled.
A signal in a *Save-signalset* is not enabled.
The saved signals are retained in the input port in the order of their arrival. The effect of the save is valid only for the state to which the save is attached. In the following state, signal instances that have been "saved" are treated as normal signal instances.
Whenever a signal is received which does not have an <input part> in the current state, it is implicitly discarded and the agent does not change its state.

Model
When the <stimulus> list of an <input part> contains more than one <stimulus>, a copy of the <input part> is created for each such <stimulus>. Then the <input part> is replaced by these copies.

3.10 Transitions

3.10.1 Free Action

Abstract grammar

Free-action	::	*Connector-name*
		Transition
Connector-name	=	*Name*

Concrete grammar

<label> ::= <connector name> :

<free action> ::= **connection** <transition> [**endconnection** [<connector name>] <end>]
"Body" refers to a state machine graph. A body encompasses <agent body>.
All the <connector name>s defined in a body must be distinct.
A label represents the entry point of a transfer of control from the corresponding joins with the same <connector name>s in the same body.
Transfer of control is only allowed to labels within the same body.
If the <transition string> of the <transition> in <free action> is non-empty, the first <action statement> must have a <label> otherwise the <terminator statement> must have a <label>.
If present, the <connector name> ending the <free action> must be the same as the <connector name> in this <label>.

Semantics
A *Free-action* defines the target of a *Join-node*. In the abstract grammar, only free actions have labels; labels inside of a transition are transformed into separate free actions.

Model
If a <label> is not the first label of a <transition string>, the <transition string> is split into two parts. All <action statements> preceding the <label> are preserved in the original transition, which is terminated with a <join> to the <label>. All action statements following <label> are copied to a new <free action>, which starts with the <label>. A similar transformation is applied if a labelled <terminator statement> is preceded by a <transition string>.

3.10.2 Transition

Abstract grammar

Transition	::	*Graph-node**
		(*Terminator* | *Decision-node*)
Graph-node	=	*Task-node*
	|	*Output-node*
	|	*Create-request-node*
	|	*Set-node*
	|	*Reset-node*
Terminator	=	*Nextstate-node*
	|	*Stop-node*
	|	*Join-node*

Concrete grammar

<transition> ::= <transition string> [<terminator statement>] | <terminator statement>

<transition string> ::= {<action statement>}+

<action statement> ::= [<label>] <action 1> <end>

<action 1> ::= <task> | <output> | <create request> | <decision> | <set> | <reset> | <export> | <import>

<terminator statement> ::= [<label>] <terminator 2> <end>

<terminator 2> ::= <nextstate> | <join> | <stop>
If the <terminator> of a <transition> is omitted, then the last action in the <transition> must contain a terminating <decision>, except when a <transition> is contained in a <decision>.

Semantics
A transition performs a sequence of actions. During a transition, the data of an agent may be manipulated and signals may be output. The transition will end with the state machine of the agent entering a state, with a stop or with the transfer of control to another transition.
A transition in one local block of a block can be interpreted at the same time as a transition in another local block of the same block or of another block.
An undefined amount of time may pass while an action is interpreted. It is valid for the time taken to vary each time the action is interpreted. It is also valid for the time taken to be the same at each interpretation or for it to be zero (that is the result of **now** is not changed).

Model
A transition action may be transformed to a list of actions (possibly containing implicit states) according to the transformation rules for <import>.

3.10.3 Terminators

Abstract grammar

Nextstate-node	::	*State-name*
Join-node	::	*Connector-name*
Stop-node	::	( )

The *State-name* specified in a nextstate must be the name of a state within the same *State-transition-graph.*
There must be exactly one *Connector-name* corresponding to a *Join-node* within the same body.

Concrete grammar

<nextstate> ::=	**nextstate** <nextstate body>
<nextstate body>::=	<state name>
<join> ::=	**join** <connector name>
<stop> ::=	**stop**

Semantics
A nextstate represents a terminator of a transition. It specifies the state of the agent when terminating the transition.
When a *Join-node* is interpreted, interpretation continues with the *Free-action* named with *Connector-name*.
The stop causes the agent interpreting it to perform a stop.
This means that the retained signals in the input port are discarded and the agent itself will cease to exist.

3.10.4 Actions

Abstract grammar

Task-node	=	*Assignment*
Create-request-node	::	*Agent-identifier*
Output-node	::	*Signal-identifier* [*Expression*]* [*Signal-destination*]
Signal-destination	=	*Expression*

In an *Output-node*, the length of the list of optional *Expression*s must be the same as the number of *Sort-name*s in the *Signal-definition* denoted by the *Signal-identifier*. Each *Expression* must be sort compatible to the corresponding (by position) *Sort-name* in the *Signal-definition.*

Concrete grammar

<task> ::=	**task** <textual task body>
<textual task body> ::=	<assignment>
<create request> ::=	**create** <create body>
<create body> ::=	<agent identifier>

<output> ::= **output** <output body>

<output body> ::= <signal identifier> [<actual parameters>]
{**,** <signal identifier> [<actual parameters>] }*
<communication constraints>

<communication constraints> ::=
[**to** <destination>]

<destination> ::= <pid expression>

<actual parameters> ::= **(** <actual parameter list> **)**

<actual parameter list> ::= [<expression>] { **,** [<expression>] }*
Commas after the last <expression> in <actual parameter list> may be omitted.
The <pid expression> in <destination> represents the *Signal-destination.*

Semantics
The interpretation of a *Task-node* is the interpretation of the *Assignment*.

The create action causes the creation of an agent instance either inside the agent that performs the create or in the agent that contains the agent that performs the create. The parent of the created agents (see Section 3.7.3 **Model**) has the same pid as returned by **self** of the creating agent. **self** of the created agents (see Section 3.7.3 **Model**) and offspring of the creating agent (see Section 3.7.3 **Model**) both have the same unique, new pid.
When an agent instance is created, it is given an empty input port, and variables are created. If the created agent has contained agent sets, then the initial instances of these sets are created in the same way. Otherwise the agent starts by interpreting the start node in the agent graph before transitions caused by signals are interpreted.
The created agent is then interpreted asynchronously and concurrently with other agents.
If an attempt is made to create more agent instances than specified by the maximum number of instances in the agent definition, then no new instance is created, the offspring of the creating agent (see Section 3.7.3 **Model**) has the result null and interpretation continues.

If no *Signal-destination* is specified in an *Output-node*, any agent for which there exists a communication path may receive the signal.
If an <expression> in <actual parameters> is omitted, no data item is conveyed with the corresponding place of the signal instance, that is, the corresponding place is "undefined".
The pid of the originating agent is also conveyed by the signal instance.
The signal instance is then delivered to a communication path able to convey it.
If *Signal-destination* is present, the signal instance is delivered to the agent instance denoted by *Signal-destination*. If this instance does not exist or is not reachable from the originating agent, the signal instance is discarded.
If no *Signal-destination* is specified, the receiver is selected in two steps. First, the signal is sent to an agent instance set, which can be reached by the communication paths able to convey the signal instance. This agent instance set is arbitrarily chosen. Second, when the signal instance arrives at the end of the communication path, it is delivered to an instance of the agent instance set. The instance is arbitrarily selected. If no instance can be selected, the signal instance is discarded.

Model
If several pairs of <signal identifier> and <actual parameters> are specified in an <output body>, this is derived syntax for specifying a sequence of <output>s in the same order as specified in the original <output body>, each containing a single pair of <signal identifier> and <actual parameters>. The **to** <destination> clause is repeated in each of the <output>s.

3.10.5 Decision

Abstract grammar

Decision-node	::	*Decision-question* *Decision-answer**-set*** [*Else-answer*]
Decision-question	=	*Expression*
Decision-answer	::	*Constant-expression**-set** Transition*
Else-answer	::	*Transition*

The *Constant-expression*s of the *Decision-answer*s must be mutually exclusive. The *Constant-expression*s of the *Decision-answer*s must be sort compatible to the sort of the *Decision-question*.

Concrete grammar

<decision> ::= **decision** <question> <end> <decision body> **enddecision**

<decision body> ::= <answer part>+ [<else part>]

<answer part> ::= **(** <answer> **)** <colon> [<transition>]

<answer> ::= <constant expression> { **,** <constant expression> } *

<else part> ::= **else** <colon> [<transition>]

<question> ::= <expression>

An <answer part> or <else part> in a decision is a terminating <answer part> or <else part> respectively if it contains a <transition> where a <terminator statement> is specified, or contains a <transition string> whose last <action statement> contains a terminating decision. A <decision> is a terminating decision, if each <answer part> and <else part> in its <decision body> is a terminating <answer part> or <else part> respectively.

Semantics

A decision transfers the interpretation to the outgoing path whose constant expression equals the result given by the interpretation of the question. A set of possible answers to the question is defined, each of them specifying the set of actions to be interpreted for that path choice.

One of the answers may be the complement of the others. This is achieved by specifying the *Else-answer*, which indicates the set of activities to be performed when the result of the expression on which the question is posed, is not covered by the results specified in the other answers.

Whenever the *Else-answer* is not specified, and the result from the evaluation of the question expression does not match one of the answers, then the further system behaviour is undefined.

Model

If a <decision> is not terminating then it is derived syntax for a <decision> wherein all not terminating <answer part>s and the <else part> if not terminating have inserted at the end of their <transition> a <join> to the first <action statement> following the decision or if the decision is the last <action statement> in a <transition string> to the following <terminator statement>.

3.10.6 Timer

Abstract grammar

Timer-definition	::	*Timer-name*
Timer-name	=	*Name*
Set-node	::	*Time-expression*
		Timer-identifier
Reset-node	::	*Timer-identifier*
Timer-identifier	=	*Identifier*
Time-expression	=	*Expression*

Concrete grammar

<timer definition> ::= **timer** <timer definition item> { **,** <timer definition item>}* <end>

<timer definition item> ::= <timer name>

<reset> ::= **reset (** <reset clause> { **,** <reset clause> }* **)**

<reset clause> ::= <timer identifier>

<set> ::= **set** <set clause> { **,** <set clause> }*

<set clause> ::= **(** <Time expression> **,** <timer identifier> **)**

A <reset clause> represents a *Reset-node*; a <set clause> represents a *Set-node*.

Semantics

A timer instance is an object, that can be active or inactive.

When an inactive timer is set, a Time value is associated with the timer. Provided there is no reset or other setting of this timer before the system time reaches this Time value, a signal with the same name as the timer is put in the input port of the agent. The same action is taken if the timer is set to a Time value less than or equal to **now**. After consumption of a timer signal the **sender** expression yields the same result as the **self** expression. A timer is active from the moment of setting up to the moment of consumption of the timer signal.

When an inactive timer is reset, it remains inactive.

When an active timer is reset, the association with the Time value is lost, if there is a corresponding retained timer signal in the input port then it is removed, and the timer becomes inactive.
When an active timer is set, this is equivalent to resetting the timer, immediately followed by setting the timer. Between this reset and set the timer remains active.
Before the first setting of a timer instance it is inactive.

Model
A <reset> or a <set> may contain several <reset clause>s or <set clause>s respectively. This is derived syntax for specifying a sequence of <reset>s or <set>s, one for each <reset clause> or <set clause> such that the original order in which they were specified in <reset> or <set> is retained.

3.11 Data

The concept of data in RSDL is defined in this clause. This includes the data terminology and the predefined data.
Data in RSDL is principally concerned with data types. A data type defines a set of elements or data items, referred to as sort, and a set of operations which can be applied to these data items. The sorts and operations define the properties of the data type. These properties are defined by data type definitions.
A data type consists of a set which is the *sort* of the data type, and one or more *operations*. As an example, consider the predefined data type Boolean. The sort Boolean of the data type Boolean consists of the elements true and false. Among the operations of the data type Boolean are "=" (equal), "/=" (not equal), "not", "and", "or", "xor", and "=>" (implies). As a further example, consider the predefined data type Integer. It has the sort Integer consisting of the elements 0, 1, -1, 2, -2, etc., and at least the operations "=", "/=", "+", "-", "*", "/", "mod", "<", ">", "<=", and ">=".
RSDL provides several predefined data types which are familiar in both their behaviour and syntax. The predefined data types are described below.
Variables are objects which can be associated with an element of a sort by assignment. When the variable is accessed, the associated data item is returned.
Operations are defined from and to elements of sorts. For instance, the application of the operation for summation ("+") from and to elements of the Integer sort is valid, whereas summation of elements of the Boolean sort is not.
Each data item belongs to exactly one sort. That is, sorts never have data items in common.
For most sorts there are literal forms to denote elements of the sort (for example, for Integers "2" is used rather than "1 + 1"). There may be more than one literal to denote the same data item (e.g. 12 and 012 can be used to denote the same Integer data item). Some sorts may have no literal forms to denote the elements of the sort; for example, the sort Time.
An expression denotes a data item. If an expression does not contain a variable or an imperative expression, e.g., if it is a literal of a given sort, each occurrence of the expression will always denote the same data item. These "passive" expressions correspond to a functional use of the language.
An expression which contains variables or imperative expressions may be interpreted as having different results during the interpretation of an RSDL system depending on the data item associated with the variables. The active use of data includes assignment to variables, use of variables, and initialization of variables. The difference between active and passive expressions is that the result of a passive expression is independent of when it is interpreted, whereas an active expression may have different results depending on the current values, or pids associated with variables or the current system state.

3.11.1 Predefined Data Types

A sort is a set of elements: values, or pids (agents identifiers). Two different sorts have no elements in common.

Abstract grammar

Literal-name	=	*Name*
Sort-name	=	*Name*

Concrete grammar

<sort> ::=	<basic sort>
<basic sort> ::=	<sort name>
<literal> ::=	<literal name>
<literal name> ::=	<literal<name>

Semantics
The following data types are predefined in RSDL: Integer, Boolean, Time, Duration, PId.
The following literals are defined for these data types.

Data type name	Literals
Integer	All <name>s of the form <decimal digit>+.
Boolean	"false", "true"
Time	None
Duration	All <name>s starting with a <decimal digit> and not being an Integer literal.
PId	"null"

Moreover, the following operations are predefined with the standard mathematical meaning.

Operation name	Comment
>=, >, <, <=	Standard comparison operators over Integer
=, /=	Standard comparison operators over all data types
+, -, *, /, mod	Standard operators over Integer
and, or, xor, =>, not	Standard operators over Boolean
+, -: Time × Duration → Time	Addition and Subtraction of Time and Duration

3.11.2 Expressions

The following subclause define how sorts, literals, and operators are interpreted in expressions.

Abstract grammar

Expression = *Constant-expression*
| *Active-expression*
Constant-expression = *Literal*
| *Operation-application*
Active-expression = *Variable-access*
| *Operation-application*
| *Imperative-expression*

Literal :: *Literal-name*

Operation-application :: *Operation-name Expression+*
Operation-name = *Name*

The *Literal-name* denotes a predefined literal.

Concrete grammar

For simplicity of description no distinction is made between the concrete syntax of *Constant-expression* and *Active-expression.*

<expression> ::= <operand>

<operand> ::= <operand0>
| <operand> <implies sign> <operand0>

<operand0> ::= <operand1>
| <operand0> { **or** | **xor** } <operand1>

<operand1> ::= <operand2>
| <operand1> **and** <operand2>

<operand2> ::= <operand3>
| <operand2>
{ <greater than sign>
| <greater than or equals sign>
| <less than sign>
| <less than or equals sign>
| <equals sign>
| <not equals sign> }
<operand3>

<operand3> ::= <operand4>
| <operand3> { <plus sign> | <hyphen> } <operand4>

<operand4> ::= <operand5>
| <operand4> { <asterisk> | <solidus> | **mod** } <operand5>

<operand5> ::= [<hyphen> | **not**] <primary>

<primary> ::= <operation application>
| <literal>
| (<expression>)
| <active primary>

<operation application> ::= <operation name> **(** <expression list> **)**

<active primary> ::= <variable access>
| <imperative expression>

<expression list> ::= <expression> { **,** <expression> }*

<constant expression> ::= <constant expression>

An <expression> which does not contain any <active primary> represents a *Constant-expression* in the abstract syntax. A <constant expression> is an <expression> that does not contain an <active primary>.
An <expression> which contains an <active primary> represents an *Active-expression*.
<operand>, <operand1>, <operand2>, <operand3>, <operand4> and <operand5> offer special syntactic forms for operation names. The special syntax is introduced, for example, so that arithmetic operations and Boolean operations can have their usual syntactic form. That is the user can write "(1 + 1) = 2" rather than being forced to use, for example, equal(add(1,1),2).
Whenever a <literal> is specified, it must denote one of the predefined literals. However, if there is a variable with the same name, the variable is used.

Semantics
An infix operator in an expression has the normal semantics of an operator but with prefix syntax.
A monadic operator in an expression has the normal semantics of an operator but with the prefix syntax.
Infix operators have an order of precedence which determines the binding of operators.
When an expression is interpreted it returns a data item (a value or pid). The returned data item is referred to as the result of the expression.
The sort of an expression is

- the sort of the <literal>, or
- the result sort of the operation, or
- the sort of the <imperative expression>, or
- the sort of the <variable access>

depending on the kind of the expression.
A *Literal* returns the unique data item defined as its value.
The sort of the <literal> is the data type it belongs to.

Model
An expression of the form
<expression> <infix operation name> <expression>
is derived syntax for
<infix operation name> **(** <expression>**,** <expression> **)**
where <infix operation name> represents an *Operation-name* although the name would not be valid concrete syntax.
Similarly,
<monadic operation name> <expression>
is derived syntax for
<monadic operation name> **(** <expression> **)**
where <monadic operation name> represents an *Operation-name*.

3.12 Variables

This subclause defines the use of data and declared variables, how an expression involving variables is interpreted, and the imperative expressions which obtain results from the underlying system.
A variable has a sort and an associated data item of that sort. The data item associated with a variable may be changed by assigning a new data item to the variable. The data item associated with the variable may be used in an expression by accessing the variable.
Any expression containing a variable is considered to be "active" since the data item obtained by interpreting the expression may vary according to the data item last assigned to the variable. The result of interpreting an active expression will depend on the current state of the system.

3.12.1 Variable Definition

A variable has a data item associated, or it is "undefined".

Abstract grammar

Variable-definition :: *Variable-name*
Sort-name
[*Constant-expression*]
Variable-name = *Name*

If the *Constant-expression* is present, it must be of the same sort as the *Sort-name* denoted.

Concrete grammar

<variable definition> ::= **dcl** [**exported**] <variables of sort> {**,** <variables of sort> }* <end>

<variables of sort> ::= <variable name> { **,** <variable name> }*
<sort> [<is assigned sign> <constant expression>]

If a <constant expression> is given in the <variable definition>, then the *Constant-expression* is represented by this <constant expression>.
Otherwise, the *Constant-expression* is not present.

Semantics

When a variable is created and the *Constant-expression* is present, then the variable is associated with the result of the *Constant-expression*. Otherwise, the variable has no data item associated, that is, the variable is "undefined".
The keyword **exported** allows a variable to be used as an exported variable as elaborated in Section 3.8.5.

Model

A <variable definition> with more than one declared variable or with more than one <variables of sort> is derived syntax for an individual variable definition for each of the variable names.

3.12.2 Variable Access

Abstract grammar

Variable-access = *Variable-identifier*

Concrete grammar

<variable access> ::= <variable identifier>

Semantics

A variable access is interpreted as giving the data item associated with the identified variable.
A variable access has a sort which is the sort of the variable identified by the variable access.
A variable access has a result which is the data item last associated with the variable. If the variable is "undefined", the further behaviour of the system is undefined.

3.12.3 Assignment

An assignment creates an association from the variable to the result of interpreting an expression.

Abstract grammar

Assignment :: *Variable-identifier*
Expression

In an *Assignment*, the sort of the *Expression* must be equal to the sort of the *Variable-identifier*.

Concrete grammar

<assignment> ::= <variable> <is assigned sign> <expression>

<variable> ::= <variable identifier>

Semantics

An *Assignment* is interpreted as creating an association from the variable identified in the assignment to the result of the expression in the assignment. The previous association of the variable is lost.

3.12.4 Imperative Expressions

Imperative expressions obtain results from the underlying system state.

Abstract grammar

Imperative-expression	=	*Now-expression*
	|	*Pid-expression*
	|	*Timer-active-expression*
Now-expression	::	( )
Pid-expression	=	*Self-expression*
	|	*Parent-expression*
	|	*Offspring-expression*
	|	*Sender-expression*
Self-expression	::	( )
Parent-expression	::	( )
Offspring-expression	::	( )
Sender-expression	::	( )
Timer-active-expression	::	*Timer-identifier*

Concrete grammar

<imperative expression> ::=
<now expression>
| <pid expression>
| <timer active expression>

<now expression> ::= **now**

<pid expression> ::= **self**
| **parent**
| **offspring**
| **sender**

<timer active expression> ::=
active (<timer identifier> **)**

Imperative expressions are expressions for accessing the system clock, the pid associated with an agent or the status of timers.

Semantics

The now expression is an expression to determine the current absolute system time.
The now expression represents an expression requesting the current value of the system clock giving the time. The origin and unit of time are system dependent. Whether two occurrences of **now** in the same transition will give the same value is system dependent. However, it always holds that:

now <= **now**;

A now expression has the Time sort.

A pid expression accesses one of the implicit anonymous variables self, parent, offspring, or sender (see Section 3.7.3 **Model**). The **self**, **parent**, **offspring** or **sender** pid expression has a result which is the last pid associated with the corresponding implicit variable as defined in Section 3.7.3.
A **self**, **parent**, **offspring**, or **sender** pid expression has a static sort which is Pid.

A timer active expression is an expression of the predefined Boolean sort which has the result true, if the timer identified by timer identifier is active (see Section 3.10.6). Otherwise the timer active expression has the result false.

3.13 Transformation of RSDL Shorthands

This clause details the transformation of the RSDL constructs, whose dynamic semantics are given after a transformation to the subset of RSDL for which an **Abstract Grammar** exists. These shorthand notations are constructs for which a **Model** section exists.
The properties of a shorthand notation are derived from the way it is modelled in terms of (or transformed to) the primitive concepts. In order to ensure easy and unambiguous use of the shorthand notations, and to reduce side effects when several shorthand notations are combined, these concepts are transformed in a specified order as detailed in this clause.

The specified order of transformation means that in the transformation of a shorthand notation of order *n*, another shorthand notation of order *m* may be used, provided *m*>*n*.
Since there is no abstract syntax for the shorthand notations, terms of concrete syntax are used in their definitions.
The transformations are described as a number of enumerated steps. One step may describe the transformation of several concepts and thus consist of a number of sub-steps, either because these concepts must be transformed as a group or because the transformation order between these concepts is not significant. The latter case is indicated by a dash (-) rather than by enumeration.

Transformation of Additional Concepts

1. Definition references are replaced by <referenced definition>s (Section 3.6.2).
2. The graphs are normalised:
   - non-terminating decisions are transformed into terminating decisions;
   - the actions and/or terminator statement following the decisions are moved to appear as <free action>s. Those generated <free action>s which have no label attached are given anonymous labels;
   - action lists (including the terminator statement which follows) where the first action (if any, otherwise the following terminator statement) has a label attached, are replaced by a join to the label and the action list appears as a <free action>.
3. Transformation of:
   - <infix operation name>s and their operands to the prefix form (Section 3.11.2);
   - State list (Section 3.9.1);
   - Multiple appearance of state is merged (Section 3.9.1).
   - Stimulus list (Section 3.9.2);
   - Multiple signals in <output body> (Section 3.10.4);
   - Multiple timers in <set> and <reset> (Section 3.10.6);
   - <channel to channel connections>, <channel definitions> by replacing/extending them with gates (Section 3.8.3);
   - Multiple variables in <variable definition> or <variables of sort> (Section 3.12.1);
   - <block definition>s are replaced by <textual typebased block definition>s (Section 3.7.3);
4. Full qualifiers are inserted:
   According to the visibility rules and the rules for resolution, qualifiers are extended to denote the full path.
5. Imported and exported values (Section 3.8.5) are transformed.

Part 4: RSDL FORMAL DEFINITION

After the informal language definition in Part 3 we will now formalise the text. We distinguish two parts of the language to be formalised, namely a static part concerning all the parts that would be handled by a compiler and the dynamic part referring to all things that are important at run time. For the static part the problem is to read in an RSDL specification as a stream of characters and to find out if the specification is correct with respect to the definition of RSDL. There are several steps to be done for this. At first, the input character string is decomposed into a sequence of tokens. This is the lexical analysis. Afterwards, the sequence of tokens is checked against the syntax structure of RSDL. After this, the syntax structure is checked against static rules, as e.g. that identifiers that are used are defined. The next step is the handling of all transformations for shorthand constructs. There is one more step, namely the transformation of the syntax structure into an abstract syntax structure, which is then used to define the dynamic semantics. See also again Figure 1 in Part 1 for an overview.

4.1 Lexis

The lexis is defined already within Section 3.4. The mathematical domain for this is BNF, with some extensions. Let us recall the task of the lexical rules. They have to provide for a unique division of the input character sequence into a sequence of tokens. For this to be possible some additional conditions have to be defined. These conditions ensure the uniqueness. These are conditions like: “A token is always the longest possible sequence of characters as defined with the lexical rules” or “When a sequence of characters could either be a keyword or a name, then it is a keyword”. These additional conditions are stated in plain natural language text, thus the lexical rules are formalised using BNF together with some more text as to be found in Section 3.4.

4.2 Syntax

The syntax is also already defined within the scope of Part 3. The mathematical domain for this is again BNF. The syntax as it is defined in Part 3 is however ambiguous. There are some places where the grammar has several alternatives which can only be distinguished using the static analysis. These places have to be unified in order to get an unambiguous grammar. An unambiguous grammar ensures that there is exactly one syntax tree representation of any correct specification.
The following changes have to be applied to the concrete syntax in order to make it unambiguous.

1. Change the production of <signal list item> to the following.
   <signal list item> ::= <identifier>
   This change is necessary because the different kinds of identifiers come into play in the static analysis.
2. Delete the alternative <literal> from the production of <primary>.
   <primary> ::= <operation application> | (<expression>) | <active primary>
   This change is necessary because <literal> is already subsumed within <variable access>. The distinction has to be done in the static analysis.

4.3 Static Semantics

The static semantics starts with a tree representation of the input as produced from step 4.2 above. For the purposes of the formal semantics definition the unambiguous concrete grammar is too verbose. So we do not use it as it is. Instead, an abstraction of it is used which is called AS0 (abstract syntax level 0). Please see below for the definition of the AS0.
In this chapter, the static semantics of RSDL is formalised. There are essentially three parts to be defined, namely the static well-formedness conditions, the transformations of the shorthand notations and the mapping to the abstract syntax AS1. For the well-formedness conditions there are two areas, namely conditions for the AS1 and conditions for the AS0. All the conditions are defined in terms of first order predicate calculus as already defined in Section 2.1.6. The context conditions are reflected in the abstract syntax tree as relations from nodes to nodes. The nodes of the AST are the objects of reasoning.

4.3.1 General Definitions

4.3.1.1 Division of Text

The static semantics is presented with the following division of text. Please find below the headings used and for each of the headings a short description of the contents.

Abstract Syntax
This part is used to describe the abstract grammar as already defined within Part 3. There will be usually no comments in this section as it is copied as is from the language definition.

Conditions on Abstract Syntax
This part reflects the conditions that can be formulated on the abstract syntax level. The conditions are usually commented by the corresponding part of the language definition.

Concrete Syntax
This part shows the concrete syntax. In fact, the abstraction of the concrete syntax, namely the AS0 as defined below, is used. There will be usually no comments in this section as it is copied from the language definition.

Auxiliary Functions
This part introduces auxiliary functions that are used later on to define the conditions on AS0 and the transformations. The aim and the definition of the functions are explained.

Conditions on Concrete Syntax
This part reflects the conditions that must be true for the concrete syntax (AS0 here). The conditions are usually commented by the corresponding part of the language definition.

Transformations
This part shows the transformations within the AS0. Please see below for the format of the rules. The transformations are usually commented by the corresponding part of the language definition.

Mapping to Abstract Syntax
This part shows how the transformed AS0 is mapped to AS1. If the mapping is straightforward, no comments are given.

4.3.1.2 Concrete Grammar (AS0)

The idea in using an abstraction of the concrete syntax is to delete the unimportant parts of the syntax tree. Therefore we distinguish precious from no-precious terminals. In our case, only <name> is precious, all the other terminal symbols stand for themselves. The use of the abstraction means also to formalise the tree structure using the abstract syntax tree representation definition as introduced within Section 2.1.6. There were two kinds of syntax rules, namely alias rules (alternatives) given as e.g.

<nt> = <nt1> | <nt2> | <nt3>

and constructor rules given as e.g.

<nt> :: <nt1> <nt2>* [<nt3>] .

Please note, that the presentation of the syntax is slightly different from the presentation in Part 3. A non terminal with an underline like <agent name> from Part 3 is presented here as <agent<name>, because all ordinary non terminals are references to their definition, and the definition in this case is <name>.
The AS0 abstraction is derived from the concrete syntax using the following abstraction rules.

1. An alternative production is mapped to an alias rule.
2. A sequence production is mapped to a constructor rule.
3. Mixed productions are resolved by introducing auxiliary rules.
4. All other productions are mapped to alias rules.
5. Delete <end> or [<end>] everywhere.
6. If one of two lexical units in a row is precious and the other one is not, delete the non precious one.
7. A non terminal followed by a sequence of the same non terminal is merged together.

See for example the abstraction of the concrete syntax of transitions (Section 3.10.2). Please note that sometimes a <keyword> or a <special> or <composite special> is not deleted because it carries information.

Concrete Syntax:

<transition> ::= {<transition string> [<terminator statement>] }
| <terminator statement>
<transition string> ::= {<action statement>}+
<action statement> ::= [<label>] { <action 1> <end> }
<action 1> ::= <task> | <output> | <create request> | <decision> | <set> | <reset> | <export> | <import>
<terminator statement> ::=[<label>] { <terminator 2> <end> }
<terminator 2> ::= <nextstate> | <join> | <stop>

AS0 Abstraction:
<transition> = <transition gen transition string> | <terminator statement>
<transition gen transition string> :: <transition string> [<terminator statement>]
<transition string> = {<action statement>}+
<action statement> :: [<label>] <action 1>
<action 1> = <task> | <output> | <create request> | <decision> | <set> | <reset> | <export> | <import>
<terminator statement> :: [<label>] <terminator 2>
<terminator 2> = <nextstate> | <join> | <stop>

4.3.1.3 Static Conditions

Usually, the AS0 conditions are checked before the transformations start. However, some conditions are only valid after some transformation steps as defined in Section 3.13. This is indicated by preceding the corresponding condition with a numbering sign (e.g. "=4=>"), where the number in the arrow indicates the next transformation step. This means, a condition with the prefix "=4=>" is checked between the transformation steps 3 and 4. By default, conditions are preceded with "=1=>", i.e. they are checked before any transformations.

4.3.1.4 Transformation Rules

Transformations are represented by rewrite rules. Please find below the syntax for rewrite rules.

<rewrite rule> ::= <pattern> "=" <integer> "=>" <expression> { **and** <dependent transformation> }*
<dependent transformation>::=
<expression> { "=>" | ":=" } <expression>

The pattern as well as the expression refer to the syntax as defined for ASM in Section 2.1. The non terminal constructor names must all match a non terminal in the concrete syntax. A variable is not allowed to appear more than once on either side. Variable names that appear on the right hand side must also appear on the left hand side. Furthermore, the pattern and expression patterns must be correctly typed and be of the same type.
A rule Pattern =i=> Expression is equivalent to an ASM rule of the form

choose v:*DefinitionAS0*
case v
Pattern1: e:=*CreateExpr*(Expression)
ReplaceIn(v.Parent, v, e)

In the definition above, *CreateExpr* means for every constructor of Expression an extend of the corresponding domain and the setting of the contents function to a corresponding **mk-** for the following sub pattern. The placeholder *ReplaceIn* means to replace v by e in the parent node of v. This does not cause problems as the syntax tree is a tree and it is always possible to find the parent and to replace one of its children.
Dependent transformation rules have a similar semantics. They are interpreted together with their main rule.
The integer in a rewrite rule means the transformation step this rule belongs to. The steps are described in Section 3.13.
We use one auxiliary function *newName* to construct new names during the transformation.

monitored *newName*: <name> → <name>

The constraint on this function is that it always returns a new unique name. However, the result is the same when the argument is the same unless the argument is *undefined*. For an *undefined* parameter a new unique name that is not already used within the syntax tree is provided.

4.3.1.5 Mapping Rules

The mapping rules in fact introduce a function

Mapping: *DefinitionAS0* → *DefinitionAS1*

The definition of the function *Mapping* is formed by the concatenation of all the cases contained in all **Mapping** sections. This is preceded with the following header part and followed by an **endcase**.

Mapping(*a*: *DefinitionAS0*): *DefinitionAS1* $=_{def}$
case *a* **of**

This way the mapping function is defined step by step in the appropriate places in the **Mapping** sections. Each alternative of the mapping will thus be preceded by a bar ("|"), because it is one alternative of the *Mapping* function description.

4.3.2 Visibility, Names and Identifiers

This section introduces all formalisations for resolution of names, visibility rules and identifiers.

4.3.2.1 Name

Abstract Syntax

Name	::	*TOKEN*
Agent-type-name	=	*Name*
Agent-name	=	*Name*
State-name	=	*Name*
Signal-name	=	*Name*
Literal-name	=	*Name*
Operation-name	=	*Name*
Timer-name	=	*Name*
Gate-name	=	*Name*
Connector-name	=	*Name*
Channel-name	=	*Name*
Variable-name	=	*Name*
Sort-name	=	*Name*

Concrete Syntax

<name> :: *TOKEN*

<literal> =<literal name>
<literal name> = <name>

<sort> = <basic sort>
<basic sort> =<name>

Mapping to Abstract Syntax

| <name>(*x*) => **mk-***Name*(*x*)

Please note that *TOKEN* is present in both AS0 and AS1 such that it need not be mapped.

4.3.2.2 Identifier

Abstract Syntax

Identifier	::	*Qualifier Name*
Qualifier	=	*Path-item**
Agent-identifier	=	*Identifier*
Agent-type-identifier	=	*Identifier*
Signal-identifier	=	*Identifier*
Timer-identifier	=	*Identifier*
Gate-identifier	=	*Identifier*
Variable-identifier	=	*Identifier*

Concrete Syntax

<identifier> :: [<qualifier>] <name>
<qualifier> :: {<path item>}+

Auxiliary Functions

For identifiers a lot of auxiliary functions are defined. They all serve for formalising the resolution, visibility and the insertion of full qualifiers. Please find below the definitions for the AS0 and AS1 grammars.

ENTITYKIND =$_{def}$
{ *agentKind*, *agentTypeKind*, *channelKind*, *signalKind*, *variableKind*, *remoteVariableKind* }

The following entity kinds exist: agents (blocks); agent types (block types); channels, gates; signals, timers; variables (including formal parameters), literals, data types; remote variables;

fullIdentifier(*i*:<identifier>): <identifier> =$_{def}$ *i.refersto0.myfullIdentifier*

The full identifier of an identifier is the full identifier of its defining entity.

fullPath(*x*: *DefinitionAS0*): <path item>* =$_{def}$
if *x* = *undefined* **then** *empty*
elseif *x* ∈ <rsdl specification> **then** *empty*
elseif *x* ∈ <block type definition> **then**
fullPath(*x.parentAS0*) ⌢ < <path item>(**block type**, *x*.**s-**<block type heading>.**s-**<name>) >
else *fullPath*(*x.parentAS0*)
endif

The full path of a definition is constructed recursively. Please note that only types may contain definitions, as instances have been transformed before.

myfullIdentifier(*x*: *DefinitionAS0*): <identifier> =$_{def}$
<identifier>(*x.fullPath*, *x.defName*)

The full identifier of a definition is constructed using its full path and its name.

myFullIdentifierAS1(*d*: *DefinitionAS1*): *DefinitionAS1* =$_{def}$
d.**inv-***Mapping.myfullIdentifier.Mapping*

The full identifier function for the AS1 is derived from that of the AS0.

resolutionByContainer(*i*:<identifier>, *k*:*ENTITYKIND*, *s*:*DefinitionAS0*):*DefinitionAS0* =$_{def}$
if *s* = *undefined* **then** *undefined*
else let *matchingDefs* =
{ *d* ∈ *DefinitionAS0*: *d.parentAS0.findScopeUnit* = *s* ∧ *i*.**s-**<name> = *d.defName* ∧
matchingQualifier(*i*.**s-**<qualifier>, *d.myfullIdentifier*.**s-**<qualifier>)} **in**
if | *matchingDefs* | = 1 **then** *matchingDefs.take*
else *resolutionByContainer*(*i*, *k*, *s.parentAS0.findScopeUnit*)
endif
endif

The binding of a <name> to a definition through resolution by container proceeds in the following steps, starting in the scope unit where the <identifier> appears:

1. if a unique entity exists in the current scope unit with the same <name> and entity kind and matching <qualifier>s, the <name> is bound to that entity; otherwise
2. resolution by container is attempted in the scope unit which defines the current scope unit.

matchingQualifier(*q1*: <qualifier>, *q2*: <qualifier>): *BOOLEAN* =$_{def}$
if *q1* = *q2* **then** *True*
elseif *q1* = *undefined* **then** *True*
elseif *q1.length* < *q2.length* **then** *matchingQualifier*(*q1*, *q2.tail*)
else *matchingPathItem*(*q1.head*, *q2.head*) ∧ *matchingQualifier*(*q1.tail*, *q2.tail*)
endif

The definition above defines when two qualifiers are matching.

refersto0(*i*: <identifier>): *DefinitionAS0* =$_{def}$
if *i.parentAS0* ∈ <typebased block heading> **then**
resolutionByContainer(*i*, *agentTypeKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ (<gate constraint> ∪ <channel path>) **then**
if *resolutionByContainer*(*i*, *signalKind*, *i.parentAS0.findScopeUnit*) ≠ *undefined* **then**
resolutionByContainer(*i*, *signalKind*, *i.parentAS0.findScopeUnit*)
else *resolutionByContainer*(*i*, *remoteVariableKind*, *i.parentAS0.findScopeUnit*)
endif
elseif *i.parentAS0* ∈ <channel endpoint> **then**
resolutionByContainer(*i*, *agentKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ <channel to channel connection> **then**
resolutionByContainer(*i*, *channelKind*, *i.parentAS0.findScopeUnit*)

elseif *i.parentAS0* ∈ <save part> **then**
resolutionByContainer(*i*, *signalKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ (<stimulus> ∪ <output body gen identifier>) **then**
resolutionByContainer(*i*, *signalKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ <assignment> **then**
resolutionByContainer(*i*, *variableKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ <create request> **then**
resolutionByContainer(*i*, *agentKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ <import> ∧ *i.parentAS0*.**s-**<identifier> = *i* **then** // it is the variable identifier
resolutionByContainer(*i*, *variableKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ <import> **then**
resolutionByContainer(*i*, *remoteVariableKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ <export> **then**
resolutionByContainer(*i*, *remoteVariableKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ (<set clause> ∪ <reset clause> ∪ <timer active expression>) **then**
resolutionByContainer(*i*, *signalKind*, *i.parentAS0.findScopeUnit*)
elseif *i.parentAS0* ∈ <operand5> **then**
if *resolutionByContainer*(*i*, *variableKind*, *i.parentAS0.findScopeUnit*) ≠ *undefined* **then**
resolutionByContainer(*i*, *variableKind*, *i.parentAS0.findScopeUnit*)
elseif *i*.**s-**<qualifier> = *undefined* **then** // it is only a <name>
predefined
else *undefined*
endif
else *undefined*
endif

The definition above lists all places where <identifier>s appear in AS0 constructors and how to resolve them using the resolution by container. Please note the special handling for <gate constraint> and <channel path> which refers to the following text in the language definition.
A <signal list item> which is an <identifier> denotes a <signal identifier> or <timer identifier> if this is possible according to the visibility rules or else a <remote variable identifier>.
Please note that the visibility rules are already covered by the definition of the resolution by context.

refersto1(*i*: *Identifier*): *DefinitionAS1* =$_{\text{def}}$
i.**inv-***Mapping.refersto0.Mapping*

The identifier reference function for the AS1 is derived from that of the AS0.

referstoName1(*n*: *Name*): *DefinitionAS1* =$_{\text{def}}$
if *n.parentAS1*∈ *Nextstate-node* **then**
let *candidateStates* =
{ *s* ∈ *parentAS1ofKind*(*n*, *State-transition-graph*).**s-***State-node*-**set**: *s*.**s-***State-name* = *n* } **in**
if | *candidateStates* | = 1 **then** *candidateStates.take* **else** *undefined* **endif**
elseif *n.parentAS1*∈ *Join-node* **then**
let *candidateLabels* =
{ *s* ∈ *parentAS1ofKind*(*n*, *State-transition-graph*).**s-***Free-action*-**set**:
s.**s-***Connector-name* = *n* } **in**
if | *candidateLabels* | = 1 **then** *candidateLabels.take* **else** *undefined* **endif**
endlet
else *undefined*
endif

The definition above shows how single names are resolved. Recall, that in some places no identifiers are allowed but only names, namely for states or connectors.

sortCompatible(*t1*: <sort>, *t2*: <sort>): *BOOLEAN* =$_{\text{def}}$ *t1* = *t2* ∧ *t1* ≠ *undefined*

The function above defines when two sorts are compatible. Due to the limited data type part an equality is sufficient.

exprSort(*e*: *Expression*): *DefinitionAS1* =$_{\text{def}}$
if *e* ∈ *Literal* **then**
mk-*Name*(*literalSort*(*e*.**s-***Name*.**s-***TOKEN*))
elseif *e* ∈ *Operation-application* **then**
mk-*Name*(*operationSort*(*e*.**s-***Operation-name*.**s-***TOKEN*,
< *exprSort*(*a*) | *a* **in** *e*.**s-***Expression*-**seq** >))
elseif *e* ∈ *Variable-access* **then**
e.refersto1.**s-***Sort-name*
elseif *e* ∈ *Now-expression* **then mk-***Name*("Time")
elseif *e* ∈ *Pid-expression* **then mk-***Name*("PId")
elseif *e* ∈ *Timer-active-expression* **then mk-***Name*("Boolean")
else *undefined*
endif
The function above calculates the sort of an expression.

findScopeUnit(*entity*: *DefinitionAS0*): *DefinitionAS0* =$_{\text{def}}$
if *entity* = *undefined* **then** *undefined*
elseif *entity* ∈ (<agent type definition> ∪ <agent definition>) **then** *entity*
else *findScopeUnit*(*entity.parentAS0*)
endif
The function *findScopeUnit* searches for the next enclosing scope unit starting from the current place.

visible(*entity*: *DefinitionAS0*, *scope*: *DefinitionAS0*): *BOOLEAN* =$_{\text{def}}$
if *scope* = *undefined* **then** *False*
else *entity.parentAS0.findScopeUnit* = *scope* ∨ *visible*(*entity*, *scope.parentAS0.findScopeUnit*)
endif
An entity is visible in a scope unit if: it has its defining context in that scope unit; or the entity is visible in the scope unit which defines that scope unit.

defName(*d*: *DefinitionAS0*): <name> =$_{\text{def}}$
if *d* ∈ <block type definition> **then** *d*.**s-**<block type heading>.**s-**<name>
elseif *d* ∈ <block definition> **then** *d*.**s-**<block heading>.**s-**<name>
elseif *d* ∈ <textual typebased block definition> **then** *d*.**s-**<typebased block heading>.**s-**<name>
elseif *d* ∈ <channel definition> **then** *d*.**s-**<name>
elseif *d* ∈ <textual gate definition> **then** *d*.**s-**<gate>
elseif *d* ∈ <signal definition> **then** *d*.**s-**<signal definition item>**-seq**.*head*.**s-**<name>
elseif *d* ∈ <timer definition> **then** *d*.**s-**<timer definition item>**-seq**.*head*
elseif *d* ∈ <variable definition> **then** *d*.**s-**<variables of sort>**-seq**.*head*.**s-**<name>**-seq**.*head*
elseif *d* ∈ <remote variable definition> **then**
d.**s-**<remote variable definition gen name>**-seq**.*head*.**s-**<name>**-seq**.*head*
else *undefined*
endif
The function *defName* extracts the name of a definition.

getEntityKind(*d*: *DefinitionAS0*): *ENTITYKIND* =$_{\text{def}}$
if *d* ∈ <agent type definition> **then** *agentTypeKind*
elseif *d* ∈ <agent type reference> **then** *agentTypeKind*
elseif *d* ∈ <agent definition> **then** *agentKind*
elseif *d* ∈ <textual typebased agent definition> **then** *agentKind*
elseif *d* ∈ <agent reference> **then** *agentKind*
elseif *d* ∈ <signal definition> **then** *signalKind*
elseif *d* ∈ <timer definition> **then** *signalKind*
elseif *d* ∈ <variable definition> **then** *variableKind*
elseif *d* ∈ <remote variable definition> **then** *remoteVariableKind*
elseif *d* ∈ <channel definition> **then** *channelKind*
elseif *d* ∈ <gate in definition> **then** *channelKind*
else *undefined*
endif

Conditions on Concrete Syntax

=5=> $\forall$ *i* $\in$ <identifier>: *i.refersto0* $\neq$ *undefined* $\wedge$ *visible*(*i.refersto0*, *i.findScopeUnit*)
An entity can be referenced by using an <identifier>, if the entity is visible.

=5=>$\forall d, d' \in$ *DefinitionAS0*: *getEntityKind*(*d*) $\neq$ *undefined* $\wedge$ *getEntityKind*(*d*) = *getEntityKind*(*d'*) $\Rightarrow$
myfullIdentifier(*d*) $\neq$ *myfullIdentifier*(*d'*)
All entities with the same entity kind must have different *Identifiers*.

Transformations

i=<identifier>(*q*, *n*) **provided** *fullIdentifier*(*i*).**s-**<qualifier> $\neq$ *q*
=4=> <identifier>(*fullIdentifier*(*i*).**s-**<qualifier>, *n*)
For all identifiers, the corresponding full qualifiers have to be inserted.

Mapping to Abstract Syntax

| *i*=<identifier>(*q*, *name*) =>
if *i.refersto0* = **predefined then mk-***Literal*(*Mapping*(*name*))
else mk-*Identifier*(*Mapping*(*q*), *Mapping*(*name*))
endif

| <qualifier>(*q*) => *Mapping*(*q*)

4.3.2.3 Path Item

Path items are mapped straightforwardly to the AS1.

Abstract Syntax

Path-item	=	*Agent-type-qualifier* |	*Agent-qualifier*
Agent-type-qualifier	::	*Agent-type-name*	
Agent-qualifier	::	*Agent-name*	

Concrete Syntax

<path item> ::<scope unit kind> <name>
<scope unit kind> = **block** | **block type**

Auxiliary Functions

matchingPathItem(*p1*: <path item>, *p2*: <path item>): *BOOLEAN* =$_{\text{def}}$
if *p1*.**s-**<scope unit kind> = **block then**
p1.**s-**<name>.*newName* = *p2*.**s-**<name> $\wedge$ *p2*.**s-**<scope unit kind> = **block type**
else *p1* = *p2*
endif
The function *matchingPathItem* provides the comparison between path items. Please note that for path items containing agent names the implicitly introduced agent type names have to be considered.

Mapping to Abstract Syntax

| <path item>(**block**, *n*) => **mk-***Agent-qualifier*(*Mapping*(*n*))
| <path item>(**block type**, *n*) => **mk-***Agent-type-qualifier*(*Mapping*(*n*))

4.3.3 General Framework

This section contains the static semantics for the specification and the referenced definitions.

4.3.3.1 RSDL Specification

Abstract Syntax

RSDL-specification	::	*Agent-type-definition Agent-definition*

Concrete Syntax

<rsdl specification> ::
<system specification> <referenced definition>*

<system specification> = <textual system specification>
<textual system specification> =
<agent definition> | <textual system specification gen agent type definition>
<textual system specification gen agent type definition> ::
<agent type definition> <textual typebased agent definition>

Mapping to Abstract Syntax

| <rsdl specification>(<textual system specification gen agent type definition>(*t*, *s*), *)
=> **mk-***RSDL-specification*(*Mapping*(*t*), *Mapping*(*s*))
In the mapping of a specification, all referenced definitions are dumped because they have been inserted into the appropriate places during the transformations (see below).

4.3.3.2 Referenced Definition

Concrete Syntax

<referenced definition> = <definition>
<definition> = <agent definition> | <agent type definition>

<agent reference> = <block reference>
<block reference> :: <block<name>

<agent type reference> = <block type reference>
<block type reference> :: <block type<name>

Auxiliary Functions

matchingRefDefs(*n*: *DefinitionAS0*): *DefinitionAS0** $=_{def}$
< *r* **in** *rootNodeAS0*.**s-**<referenced definition>:
((*n*.*getEntityKind* = *r*.*getEntityKind*) ∧ (*n*.**s-**<name> = *r*.**s-**<name>)) >
The function *matchingRefDefs* calculates a set of all matching referenced definitions for one name.

references(*n*: *DefinitionAS0*): *DefinitionAS0* $=_{def}$ *matchingRefDefs*(*n*).*head*
The function *references* extracts an element of all matching referenced definitions. By the conditions below it is ensured that there is only one.

referencedBy(*n*: *DefinitionAS0*): *DefinitionAS0*-**set** $=_{def}$
{ *r* ∈ *DefinitionAS0*: *references*(*r*) = *n* }
The function *referencedBy* calculates the set of all definition references that use a referenced definition.

Conditions on Concrete Syntax

∀ *r* ∈ (<agent reference> ∪ <agent type reference>): *r*.*matchingRefDefs*.*length* = 1

∀ *n* ∈ <referenced definition>: | *referencedBy*(*n*) | = 1
For each <referenced definition> there must be a reference in the associated <system specification>.
A <name> is present in a <referenced definition> after the initial keyword(s). For each reference there must exist exactly one <referenced definition> with the same <name> and entity kind as the reference.

Transformations

r = <block reference>(*) =1=> *references*(*r*)
r = <block type reference>(*) =1=> *references*(*r*)
The transformation just says how to include the referenced definition into the place where it belongs. Deletion of the definition is done when the enclosing construct is transformed (i.e. when transforming <rsdl specification>).

4.3.4 Agents

This section formalises all aspects of agents, name type definitions, typebased definitions and direct definitions.

4.3.4.1 Agent Type Definitions

Abstract Syntax

Agent-type-definition :: *Agent-type-name*
Signal-definition-**set**
Timer-definition-**set**
Variable-definition-**set**
Agent-type-definition-**set**
Agent-definition-**set**
Gate-definition-**set**
Channel-definition-**set**
[*State-transition-graph*]

Conditions on Abstract Syntax

$\forall\ a \in$ *Agent-type-definition*: *a*.**s-***Agent-definition*-**set** $\neq \emptyset \Rightarrow$
a.**s-***Variable-definition*-**set** $= \emptyset \wedge$ *a*.**s-***State-transition-graph* = *undefined*

If *Agent-definition*-**set** is not empty, *Variable-definition*-**set** must be empty and *State-transition-graph* must not be present.

$\forall\ a \in$ *Agent-type-definition*: *a*.**s-***State-transition-graph* $\neq$ *undefined* $\Rightarrow$
a.**s-***Agent-definition*-**set** = *empty* $\wedge$ *a*.**s-***Channel-definition*-**set** = *empty*

If *State-transition-graph* is present, *Agent-definition*-**set** and *Channel-definition*-**set** must be empty.

Concrete Syntax

<agent type definition> = <block type definition>

<block type definition> ::
<block type heading> <agent type structure> [<block type<name>]
<block type heading> :: <block type<name>

<agent type structure> ::
{ <entity in agent>
| <channel definition>
| <channel to channel connection>
| <gate in definition>
| <agent definition>
| <agent reference>
| <textual typebased agent definition>}*
[<agent type body>]

<entity in agent> =
<signal definition>
| <variable definition>
| <remote variable definition>
| <timer definition>
| <agent type definition>
| <agent type reference>

<agent type body> = <state machine graph>

Conditions on Concrete Syntax

∀ *b* ∈ <block type definition>:
b.**s-**<name> ≠ *undefined* ⇒ *b*.**s-**<name> = *b*.**s-**<block type heading>.**s-**<name>

The optional name in a definition after the ending keyword must be syntactically the same as the name following the commencing keyword.

Mapping to Abstract Syntax

| <block type definition>(<block type heading>(*name*), <agent type structure>(*entities*, *body*), *)
=> **mk-***Agent-type-definition*(*Mapping*(*name*),
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Signal-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Timer-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Variable-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Agent-type-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Agent-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Gate-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Channel-definition*) },
Mapping(*body*))

Within the mapping, the corresponding entities are selected and filled into the lists of the abstract syntax tree. For example the signal definitions are selected as < *e* in *Mapping*(*entities*): *e* ∈ *Signal-definition* > meaning: select from the sequence *Mapping*(*entities*) all elements *e* with the condition *e* ∈ *Signal-definition*.

4.3.4.2 Agent Type Based Definitions

Abstract Syntax

Agent-definition	::	*Agent-name* *Number-of-instances* *Agent-type-identifier*

Conditions on Abstract Syntax

∀*d* ∈ *Agent-definition*:
parentAS1(*d*) ∈ *RSDL-specification* ⇒
(*d*.**s-***Number-of-instances*.**s-***Initial-number* = 1) ∧
(*d*.**s-***Number-of-instances*.**s-***Maximum-number* = 1)

In the outermost *Agent* the *Initial-number* of instances is 1 and the *Maximum-number* of instances is 1.

Concrete Syntax

<textual typebased agent definition> = <textual typebased block definition>
<textual typebased block definition> :: <typebased block heading>

<typebased block heading> ::
<block><name> [<number of instances>] <block><type expression>

<type expression> = <base type>
<base type> =<identifier>

Transformations

<typebased block heading>(*n*, *undefined*, *t*) =3=>
<typebased block heading>(*n*, <number of instances>(*undefined*, *undefined*), *t*)

A missing number of instances is replaced by an empty one, which is (partly) filled later, see Section 4.3.4.4.

Mapping to Abstract Syntax

| <textual typebased block definition>(<typebased block heading>(*n*, *inst*, *b*))
=> **mk-***Agent-definition*(*Mapping*(*n*), *Mapping*(*inst*), *Mapping*(*b*))

4.3.4.3 Direct Agent Definitions

Concrete Syntax

<agent structure> = <agent type structure>
<agent definition> = <block definition>
<block definition> :: <block heading> <agent structure> [<block<name>]

<block heading> :: <block<name> <agent instantiation>
<agent instantiation> = [<number of instances>]

Conditions on Concrete Syntax

∀ *b* ∈ <block definition>:
b.**s-**<name> ≠ *undefined* ⇒ *b*.**s-**<name> = *b*.**s-**<block heading>.**s-**<name>

The optional name in a definition after the ending keyword must be syntactically the same as the name following the commencing keyword.

Transformations

< *b*=<block definition>(<block heading>(*n*, *inst*), *structure*, *) >
=3=> < <block type definition>(<block type heading>(*newName*(*n*)), *structure*, *undefined*),
<textual typebased block definition>(
<typebased block heading>(*n*, *inst*, <identifier>(*fullPath*(*b*), *newName*(*n*)))) >

<rsdl specification>(*b*=<block definition>(<block heading>(*n*, *inst*), *structure*, *), *refs*)
=3=> <rsdl specification>(<textual system specification gen agent type definition>(
<block type definition>(<block type heading>(*newName*(*n*)), *structure*, *undefined*),
<textual typebased block definition>(
<typebased block heading>(*n*, *inst*, <identifier>(*fullPath*(*b*), *newName*(*n*))))), *refs*)

An *Agent-definition* has an implied anonymous agent type that defines the properties of the agent.
The transformation rule above describes how the implicit agent type is created. The <block definition> is replaced by a <block type definition> followed by a <textual typebased block definition> referring to the newly introduced type.

4.3.4.4 Number of Instances

Abstract Syntax

Number-of-instances	::	*Initial-number* [*Maximum-number*]
Initial-number	=	*INT*
Maximum-number	=	*INT*

Conditions on Abstract Syntax

∀*n* ∈ *Number-of-instances*: *n*.**s-***Maximum-number* ≠ *undefined* ⇒
n.**s-***Initial-number* ≤ *n*.**s-***Maximum-number* ∧ *n*.**s-***Maximum-number* > 0

The <initial number> of instances must be less than or equal to <maximum number> and <maximum number> must be greater than zero.

Concrete Syntax

<number of instances> :: [<initial number>] [<maximum number>]
<initial number> = <Integer<name>
<maximum number> = <Integer<name>

Conditions on Concrete Syntax

∀ *n* ∈ <number of instances>: *n*.**s-**<initial number> ≠ *undefined* ⇒
n.**s-**<initial number>.**s-***TOKEN.isIntToken*

The initial number must be of type Integer.

∀ *n* ∈ <number of instances>: *n*.**s-**<maximum number> ≠ *undefined* ⇒
n.**s-**<maximum number>.**s-***TOKEN.isIntToken*
The maximum number must be of type Integer.

Transformations

<number of instances>(*undefined*, *max*) =3=> <number of instances>(<name>("1"), *max*)
If <initial number> is omitted, then <initial number> is 1.

n=<number of instances>(*ini*, *undefined*)
provided *n.parentAS0.parentAS0.parentAS0* ∈ <rsdl specification>
=3=> <number of instances>(*ini*, <name>("1"))
In the outermost *Agent* the *Maximum-number* of instances is 1.

Mapping to Abstract Syntax

| <number of instances>(*i*, *m*) =>
mk-*Number-of-instances*(*i*.**s-***TOKEN.getIntValue*,
if *m*= *undefined* **then** *undefined* **else** *m*.**s-***TOKEN.getIntValue* **endif**)

4.3.5 Variables

Abstract Syntax

Variable-definition	::	*Variable-name* *Sort-name* [*Constant-expression*]

Conditions on Abstract Syntax

∀*d* ∈ *Variable-definition*: **s-***Constant-expression*(*d*) ≠ *undefined* ⇒
sortCompatible(*exprSort*(**s-***Constant-expression*(*d*)), **s-***Sort-name*(*d*))
If the *Constant-expression* is present, it must be of the same sort as the one denoted by *Sort-name*.

Concrete Syntax

<variable definition> :: [**exported**] {<variables of sort>}+
<variables of sort> :: <variable<name>+ <sort> [<constant expression>]

Auxiliary Functions

myImplicitVariableName(*v*: <variable definition>): <name> $=_{def}$
<name>("imc" ⌢ *v*.**s-**<variables of sort>**-seq**.*head*.**s-**<name>**-seq**.*head*.**s-***TOKEN*)
We introduce a function for the implicit variable name of exported variables.

controlled *statesInserted*: <variable definition> → *BOOLEAN*
We introduce a controlled function for remembering if the remote variable handling was already inserted into the states.

Transformations

< <variables of sort>(< *n* > ⌢ *r*, *s*, *e*) > **provided** *r* ≠ *empty*
=3=> < <variables of sort>(< *n* >, *s*, *e*), <variables of sort>(*r*, *s*, *e*) >
Multiple variable definitions are separated.

< <variable definition>(*e*, < *v* > ⌢ *r*) > **provided** *r* ≠ *empty*
=3=> < <variable definition>(*e*, < *v* >), <variable definition>(*e*, *r*) >
Multiple variable definitions are separated.

v=<variable definition>(**exported**, < <variables of sort>(< *n* >, *s*, *e*) >)
provided *myImplicitVariableName*(*v*).*refersto0 = undefined*
=5=> < *v*, <variable definition>(*undefined*,
< <variables of sort>(< *myImplicitVariableName*(*v*) >, *s*, *e*) >) >

For every exported variable, an implicit variable is defined.

v=<variable definition>(**exported**, *) **provided** *statesInserted*(*v*) = *False*
=5=> *v*
and *statesInserted*(*v*) := *True*
and do forall *s*: *s* ∈ *v.parentAS0*.**s-**<agent type body>.**s-implicit** ∧ *s* ∈ <basic state>
let *newInput* = <input part>(< <stimulus>(*v.myQuerySignalIdentifier*, *empty*) >,
< <action statement>(<output>(<output body>(
< <output body gen identifier>(*v.refersto0.myReplySignalIdentifier*,
<actual parameters>(< <identifier>(*v.fullPath*, *v.myImplicitVariableName*) >)) >,
<operand5>(*undefined*, **sender**)))) >,
<terminator statement>(*undefined*, <nextstate>(*s*.**s-**<state list>.*head*)))
in
=> <basic state>(*s*.**s-**<state list>, *s*.**s-implicit** ⌢ *newInput*, *s*.**s-**<name>)

To all <state>s of the exporter, excluding implicit states derived from import, the following <input part> is added:

input xQUERY;
output xREPLY(imcx) **to sender**;
nextstate *the state containing this input*;

imcx denotes the implicit copy of the exported variable.
The above rule actually states three parallel transformations: First, the variable definition *v* is replaced by itself. Additionally, the value of *statesInserted*(*v*) is set to *True*. Moreover, all states within the enclosing agent structure are modified.

Mapping to Abstract Syntax

| <variable definition>(*, < *var* >) => *Mapping*(*var*)
| <variables of sort>(< *name* >, *sort*, *const*)
=> **mk-***Variable-definition*(*Mapping*(*name*), *Mapping*(*sort*), *Mapping*(*const*))

4.3.6 Communication

4.3.6.1 Signal

Abstract Syntax

Signal-definition :: *Signal-name Sort-name**

Concrete Syntax

<signal definition>:: <signal definition item>+
<signal definition item> :: <signal<name> [<sort list>]
<sort list> :: {<sort>}+

Transformations

< <signal definition>(< *i* > ⌢ *r*) > **provided** *r* ≠ *empty*
=3=> < <signal definition>(< *i* >), <signal definition>(*r*) >

Multiple signal definitions are separated.

Mapping to Abstract Syntax

| <signal definition>(< <signal definition item>(*name*, *sorts*) >)
=> **mk-***Signal-definition*(*Mapping*(*name*),
if *sorts*=*undefined* **then** *empty* **else** *Mapping*(*sorts*) **endif**)
| <sort list>(*sorts*) => *Mapping*(*sorts*)

4.3.6.2 Gate

Abstract Syntax

Gate-definition	::	*Gate-name* *In-signal-identifier*-**set** *Out-signal-identifier*-**set**
In-signal-identifier	=	*Signal-identifier*
Out-signal-identifier	=	*Signal-identifier*

Concrete Syntax

<gate in definition> = <textual gate definition>
<textual gate definition> :: <gate> <gate constraint> [<gate constraint>]

<gate> = <gate<name>

<gate constraint> :: { **out** | **in** } <signal list>

<signal list> = {<signal list item>}+
<signal list item> = <identifier>

Auxiliary Functions

findSignalset(*c*: <gate constraint>, *t*: *TOKEN*): *DefinitionAS0** $=_{def}$
if *c*=*undefined* **then** *empty*
elseif *c*.**s-implicit** = *t* **then** *c*.**s-**<signal list>
else *empty*
endif

We introduce an auxiliary function to extract the signal list per direction.

Conditions on Concrete Syntax

$\forall$ *g* $\in$ <textual gate definition>: *g*.**s2-**<gate constraint> $\neq$ *undefined* $\Rightarrow$
g.**s-**<gate constraint>.**s-implicit** $\neq$ *g*.**s2-**<gate constraint>.**s-implicit**

Where two <gate constraint>s are specified one must be in the reverse direction to the other.

Mapping to Abstract Syntax

| <textual gate definition>(*g*, *c1*, *c2*) =>
mk-*Gate-definition*(*Mapping*(*g*),
Mapping(*findSignalset*(*c1*, **in**)).*toSet* $\cup$ *Mapping*(*findSignalset*(*c2*, **in**)).*toSet*,
Mapping(*findSignalset*(*c1*, **out**)).*toSet* $\cup$ *Mapping*(*findSignalset*(*c2*, **out**)).*toSet*)

In the mapping to the AS1 the gate constraints must be extracted per direction. See the definition of *findSignalset* above.

4.3.6.3 Channel Definition

Abstract Syntax

Channel-definition	::	*Channel-name* *Channel-path*-**set**

Conditions on Abstract Syntax

$\forall$ *p* $\in$ *Channel-path*:
(*p*.**s-***Originating-gate.refersto1.parentAS1* = *p*.**s-***Destination-gate.refersto1.parentAS1*) $\Rightarrow$
| *p.parentAS1*.**s-***Channel-path*-**set** | = 1

If the *Originating-gate* and the *Destination-gate* are in the same *Agent-definition*, the channel must be unidirectional (i.e., the second *Channel-path* in *Channel-definition* must be absent).

$\forall p_1 \in$ *Channel-path*: $\forall p_2 \in$ *Channel-path*: $p_1 \neq p_2 \wedge$ *parentAS1*(p_1) = *parentAS1*$(p_2) \Rightarrow$
s-*Originating-gate*(p_1) = **s-***Destination-gate*$(p_2) \wedge$
s-*Originating-gate*(p_2) = **s-***Destination-gate*(p_1)

When there are two paths the channel is bi-directional and the *Originating-gate* of each *Channel-path* must be the same as the *Destination-gate* of the other *Channel-path*.

Concrete Syntax

<channel definition> ::
[<channel<name>] <channel path> [<channel path>] [<channel<name>]

Conditions on Concrete Syntax

$\forall c \in$ <channel definition>: *c*.**s-**<name> = *undefined* $\Rightarrow$ *c*.**s2-**<name> = *undefined*

The ending <channel name> may only be specified if the starting <channel name> is specified.

$\forall c \in$ <channel definition>: *c*.**s2-**<name> $\neq$ *undefined* $\Rightarrow$ *c*.**s-**<name> = *c*.**s2-**<name>

The optional name in a definition after the ending keyword must be syntactically the same as the name following the commencing keyword.

Transformations

<channel definition>(*undefined*, *p1*, *p2*, *undefined*)
=3=> <channel definition>(*newName*(*undefined*), *p1*, *p2*, *undefined*)

If the <channel name> is omitted from a <channel definition>, the channel is implicitly and uniquely named.

Mapping to Abstract Syntax

| <channel definition>(*n*, *p1*, *p2*, *)
=> **mk-***Channel-definition*(*Mapping*(*n*),
if *p2=undefined* **then** {*Mapping*(*p1*)} **else** {*Mapping*(*p1*), *Mapping*(*p2*)} **endif**)

The mapping means just to construct the corresponding channel definition from the paths available.

4.3.6.4 Channel Path

Abstract Syntax

Channel-path	::	*Originating-gate*
		Destination-gate
		*Signal-identifier***-set**
Originating-gate	=	*Gate-identifier*
Destination-gate	=	*Gate-identifier*

Conditions on Abstract Syntax

$\forall p \in$ *Channel-path*:
p.**s-***Originating-gate.refersto1.parentAS1* = *p.parentAS1.parentAS1* $\vee$
p.**s-***Destination-gate.refersto1.parentAS1* = *p.parentAS1.parentAS1*

The *Originating-gate* or *Destination-gate* must be defined in the same scope unit in the abstract syntax in which the channel is defined.

Concrete Syntax

<channel path> :: <channel endpoint> <channel endpoint> <signal list>
<channel endpoint> :: {<agent<identifier> | **env**} [<via gate>]
<via gate> :: <gate>

Conditions on Concrete Syntax

$\forall c \in$ <channel endpoint>: *c*.**s-**<via gate> $\neq$ *undefined* $\Rightarrow$
findconnect(*c.parentAS0.parentAS0.parentAS0*, *c*.**s-implicit**) $\neq$ *undefined*

Every channel must be connected to somewhere. The definition of findconnect can be found in Section 4.3.6.5.

Transformations

c=<channel endpoint>(*id*, *undefined*) **provided** *findconnect*(*c.parentAS0.parentAS0*, *id*) ≠ *undefined*
=3=> <channel endpoint>(*id*, *findconnect*(*c.parentAS0.parentAS0*, *id*))

In the surrounding scope unit the <channel definition> that is identified by the <channel identifier> is extended with a <via gate> part. The <via gate> part is added to the <channel endpoint> that references the current scope unit and it mentions the implicit gate. Inside the scope unit the channels that are associated with the external channel by means of the <channel to channel connection> are modified, by extending the <channel endpoint> that mentions **env** with a <via gate> part for the implicit gate.

c=<channel endpoint>(*id*, *undefined*)
provided *findconnect*(*c.parentAS0.parentAS0*, *id*) = *undefined* ∧
c.parentAS0.parentAS0.parentAS0.parentAS0 ∈ <rsdl specification>
=3=> *c*
and
let *d* = *c.parentAS0.parentAS0* **in**
< *d* > => < *d*, <channel to channel connection>(*empty*, < *d.myfullIdentifier* >) >

An implicit gate is also introduced for channels going to the environment of the system.

Mapping to Abstract Syntax

| <channel path>(*endp1*, *endp2*, *with*)
=> **mk-***Channel-path*(*Mapping*(*endp1*), *Mapping*(*endp2*),
if *with*=*undefined* **then** ∅ **else** *Mapping*(*with*))
| <channel endpoint>(*, *gate*) => *Mapping*(*gate*)
| <via gate>(*gate*) => *Mapping*(*gate*)

4.3.6.5 Connections

Concrete Syntax

<channel to channel connection> ::
<external channel identifiers> <channel identifiers>
<external channel identifiers> = {<channel<identifier>}+
<channel identifiers> = {<channel<identifier>}+

Auxiliary Functions

controlled *myImplicitGateIdentifier*: <channel to channel connection> → <identifier>

We introduce an auxiliary function to store the implicitly generated gate identifier of a connection.

findconnect(*ch*:<channel definition>, *id*: *DefinitionAS0*): <identifier> =$_{def}$
if *id*=**env then**
let *matchingGateIds* =
{ *c.myImplicitGateIdentifier* | *c* ∈ <channel to channel connection>:
c.parentAS0 = *ch.parentAS0* ∧ *myfullIdentifier*(*c*) ∈ *c*.**s-**<channel identifiers>} **in**
matchingGateIds.take
else
let *matchingGateIds* =
{ *c.myImplicitGateIdentifier* | *c* ∈ <channel to channel connection>:
c.parentAS0 = *id.refersto0* ∧ *myfullIdentifier*(*c*) ∈ *c*.**s-**<external channel identifiers>} **in**
matchingGateIds.take
endif

The function *findconnect* computes the implicit gate identifier for a channel that is mentioned in a channel to channel connection.

bigSeq(*s*: *DefinitionAS0***): *DefinitionAS0** =$_{def}$
if *s* = *empty* **then** *empty* **else** *s.head* ⌢ *s.tail.bigSeq*

The above function serves to concatenate the elements of a sequence of sequences.

allSignalsIn(*c*: <channel to channel connection>): *DefinitionAS0** =$_{def}$
bigSeq(< *id*.*refersto0*.**s-**<channel path>.**s-**<signal list> | *id* **in** *c*.**s-**<channel identifiers> >) $^\frown$
bigSeq(< *id*.*refersto0*.**s-**<channel path>.**s-**<signal list> | *id* **in** *c*.**s-**<external channel identifiers> >)
The above function computes the input signals belonging to a channel to channel connection.

allSignalsOut(*c*: <channel to channel connection>): *DefinitionAS0** =$_{def}$
bigSeq(< *id*.*refersto0*.**s-**<channel path>.**s-**<signal list> | *id* **in** *c*.**s-**<channel identifiers> >) $^\frown$
bigSeq(< *id*.*refersto0*.**s-**<channel path>.**s-**<signal list> | *id* **in** *c*.**s-**<external channel identifiers> >)
The above function computes the output signals belonging to a channel to channel connection.

Conditions on Concrete Syntax

$\forall$ *c* $\in$ <channel to channel connection>:
| *c*.**s-**<external channel identifiers>.*toSet* | = *c*.**s-**<external channel identifiers>.*length* $\wedge$
| *c*.**s-**<channel identifiers>.*toSet* | = *c*.**s-**<channel identifiers>.*length*

$\forall$ *c1*, *c2* $\in$ <channel to channel connection>:
c1.*parentAS0* = *c2*.*parentAS0* $\Rightarrow$
c1.**s-**<external channel identifiers>.*toSet* $\cap$ *c2*.**s-**<external channel identifiers>.*toSet* = $\varnothing$ $\wedge$
c1.**s-**<channel identifiers>.*toSet* $\cap$ *c2*.**s-**<channel identifiers>.*toSet* = $\varnothing$
No channel may be mentioned after the keyword **and** in more than one <channel to channel connection> of a given scope unit. No channel may be mentioned before the keyword **and** in more than one <channel to channel connection> of a given scope unit.
The first condition above states that no sequence of channels contains duplicate elements, the second one deals with the relations between different connections.

Transformations

let *nn*=*newName*(*undefined*) **in**
< *c*=<channel to channel connection>(*, *) > **provided** *c*.*myImplicitGateIdentifier* = *undefined*
=3=> < *c*, <textual gate definition>(*nn*,
<gate constraint>(**out**, *allSignalsOut*(*c*)), <gate constraint>(**in**, *allSignalsIn*(*c*))) >
and
c.*myImplicitGateIdentifier*:= <identifier>(*fullPath*(*c*), *nn*)
Each different <channel to channel connection> in a given scope unit defines one implicit gate on the scope unit. All channels in the <channel to channel connection> are connected to that gate in their respective scope units. The gate constraints of the implicit gate are derived from the channels connected to the gate.
The name of the gate is a unique and unambiguous derived name.

4.3.6.6 Timer

Abstract Syntax

Timer-definition :: *Timer-name*

Concrete Syntax

<timer definition> :: <timer definition item>+
<timer definition item> = <timer<name>

Transformations

< <timer definition>(< *i* > $^\frown$ *r*) > **provided** *r* $\neq$ *empty*
=3=> < <timer definition>(< *i* >), <timer definition>(*r*) >
Multiple timer definitions are separated.

Mapping to Abstract Syntax

| <timer definition>(< *n* >)
=> **mk-***Timer-definition*(*Mapping*(*n*))

4.3.6.7 Remote Variable Definition

Concrete Syntax

<remote variable definition> :: <remote variable definition gen name>+
<remote variable definition gen name> ::
<remote variable<name>+ <sort>
<import> ::
<variable> <remote variable<identifier> <communication constraints>
<export> :: <variable<identifier>+

Auxiliary Functions

myQuerySignalIdentifier(*r*: <remote variable definition>): <identifier> =$_{def}$
<identifier>(*fullPath*(*r*), *r*.**s-**<remote variable definition gen name>.**s-**<name> ⌢ "Query")

myReplySignalIdentifier(*r*: <remote variable definition>): <identifier> =$_{def}$
<identifier>(*fullPath*(*r*), *r*.**s-**<remote variable definition gen name>.**s-**<name> ⌢ "Reply")

We define two functions to find the implicit query and reply signal names for a remote variable.

completeInputSet(*a*: <agent type definition>): *DefinitionAS0** =$_{def}$
bigSeq(< (*findSignalset*(*g*.**s-**<gate constraint>, **in**) ⌢ *findSignalset*(*g*.**s2-**<gate constraint>, **in**)) |
g **in** *a*.**s-**<agent type structure>.**s-implicit**: (*g* ∈ <gate in definition>) >)

The function *completeInputSet* computes the set of all signals that are allowed at the gates of the agent.

Conditions on Concrete Syntax

∀ *e* ∈ <export>:
e.**s-**<identifier>**-seq**.*head*.*refersto0* ∈ <remote variable definition>

An <identifier> in an <export> refers to a remote variable.

∀ *i* ∈ <import>:
i.**s-**<variable>.*refersto0* ∈ <variable definition>

A <variable> in an <import> refers to a variable.

∀ *i* ∈ <import>:
i.**s-**<identifier>.*refersto0* ∈ <remote variable definition>

An <identifier> in an <import> refers to a remote variable.

Transformations

< <remote variable definition gen name>(< *n* > ⌢ *r*, *s*) > **provided** *r* ≠ *empty*
=3=> < <remote variable definition gen name>(< *n* >, *s*),
<remote variable definition gen name>(*r*, *s*) >

Multiple remote variable definitions are separated.

< <remote variable definition>(< *v* > ⌢ *r*) > **provided** *r* ≠ *empty*
=3=> < <remote variable definition>(< *v* >), <remote variable definition>(*r*) >

Multiple remote variable definitions are separated.

< *r*=<remote variable definition>(< <remote variable definition gen name>(< *n* >, *s*) >) >
provided *myQuerySignalIdentifier*(*r*).*refersto0* = *undefined*
=5=> < *r*,
<signal definition>(< <signal definition item>(*myQuerySignalIdentifier*(*r*), *empty*) >),
<signal definition>(< <signal definition item>(*myReplySignalIdentifier*(*r*), < *s* >) >) >

The implicit variables are declared.

i=<identifier>(*p*,*n*)
provided *parentAS0*(*i*) ∈ <channel path> ∧ *i.refersto0* ∈ <remote variable definition>
=5=> *i.refersto0.myQuerySignalIdentifier*
and let *nn*=*newName*(*undefined*) **in**
let *c*=*i.parentAS0.parentAS0* **in**
< *c* >
=> < *c*,
<channel definition>(*nn*,
<channel path>(*i.parentAS0*.**s2-**<channel endpoint>, *i.parentAS0*.**s-**<channel endpoint>,
< *i.refersto0.myReplySignalIdentifier* >),
undefined, *undefined*) >

The query signal identifier is inserted into the channels and a new channel for the reply is created.

<export>(< *i* > ⌢ *r*) **provided** *r* ≠ *empty*
=3=> < <export>(< *i* >), <export>(*r*) >

Multiple exports are separated.

<export>(< *i* >)
=5=> <assignment>(*i.refersto0.myImplicitVariableName*, <operand5>(*undefined*, *i*))

An export is replaced by copying into the implicit variable.

let *nn*=*newName*(*undefined*) **in**
< *i*=<import>(*v*, *id*, *constr*) > ⌢ *r*
=5=> < <output>(<output body>(
< <output body gen identifier>(*id.refersto0.myQuerySignalIdentifier*, *empty*) >, *constr*)) >
and
i.parentAS0 => <transition gen transition string>(*i.parentAS0*.**s-**<action statement>,
<terminator statement>(*undefined*, <nextstate>(*nn*)))
and let *h*=*parentAS0ofKind*(*i*, <basic state>) **in**
< *h* >
=> < *h*,
<basic state>(< *nn* >,
< <save part>(*parentAS0ofKind*(*h*, <agent type definition>).*completeInputSet*),
<input part>(< <stimulus>(*i.refersto0.myReplySignalIdentifier*, < *v* >) >, *r*)
>, *undefined*) >

The <import>
v:= **import** (x, **to** destination)
is transformed to the following, where the **to** clause is omitted if the destination is not present in the original expression:
output xQUERY **to** destination;
wait in state xWAIT, saving all other signals;
input xREPLY(v);

Mapping to Abstract Syntax

| < <remote variable definition>(*) > => *empty*

The remote variable definition is ignored in the mapping.

4.3.7 State Machine

4.3.7.1 State Transition Graph

Abstract Syntax

State-transition-graph	::	*Start-node*
		State-node-**set**
		Free-action-**set**

Conditions on Abstract Syntax

∀ *g* ∈ *State-transition-graph*:
| *g*.**s-***Free-action***-set** | = | { *f*.**s-***Connector-name* | *f* ∈ *g*.**s-***Free-action***-set** } |
All the <connector name>s defined in a body must be distinct.

Concrete Syntax

<state machine graph> ::
<start> {<state> | <free action>}*

Mapping to Abstract Syntax

| <state machine graph>(*st*, *items*) =>
mk-*State-transition-graph*(*Mapping*(*st*),
{ *i* ∈ *Mapping*(*items*).*toSet*: *i* ∈ *State-node*},
{ *i* ∈ *Mapping*(*items*).*toSet*: *i* ∈ *Free-action*})

4.3.7.2 Start Node

Abstract Syntax

Start-node	::	*Transition*

Concrete Syntax

<start> :: <transition>

Mapping to Abstract Syntax

| <start>(*trans*) => **mk-***Start-node*(*Mapping*(*trans*))

4.3.7.3 State Node

Abstract Syntax

State-node	::	*State-name* *Save-signalset* *Input-node***-set** *Continuous-signal***-set**
Save-signalset	=	*Signal-identifier***-set**

Conditions on Abstract Syntax

∀ sn_1, sn_2 ∈ *State-node*: ($sn_1 \neq sn_2$) ∧ (*parentAS1*(sn_1)=*parentAS1*(sn_2)) ⇒
(**s-***State-name*(sn_1) ≠ **s-***State-name*(sn_2))
*State-node*s within a *State-transition-graph* must have different *State-name*.

Concrete Syntax

<state> = <basic state>
<basic state> ::
<state list>
{ <input part> | <save part> | <continuous signal>}*
[<state<name>]
<state list> = <state<name>+
<save part> :: <save list>
<save list> = <signal list>

Conditions on Concrete Syntax

∀ *s* ∈ <basic state>: *length*(*s*.**s-**<state list>) ≠ 1 ⇒ *s*.**s-**<name> = *undefined*
The optional <state name> ending a <state> may be specified only if the <state list> in the <state> consists of a single <state name> in which case it must be that <state name>.

∀ *s* ∈ <basic state>: *s*.**s-**<name> ≠ *undefined* ⇒ *s*.**s-**<name> = *s*.**s-**<state list>.*head*
The optional name in a definition after the ending keyword must be syntactically the same as the name following the commencing keyword.

Transformations

< <basic state>(< *n* > ⌢ *r*, *t*, *) > **provided** *r* ≠ *empty*
=3=> < <basic state>(< *n* >, *t*, *undefined*), <basic state>(*r*, *t*, *undefined*) >
Multiple basic states are separated.

< *b1*=<basic state>(< *n* >, *t1*, *) > ⌢ *x* ⌢ < *b2*=<basic state>(< *n* >, *t2*, *) >
=3=> < <basic state>(< *n* >, *t1* ⌢ *t2*, *undefined*) > ⌢ *x*
Basic states with the same name are merged.

< <save part>(< *s* > ⌢ *r*) > **provided** *r* ≠ *empty*
=3=> < <save part>(< *s* >), <save part>(*r*) >
Multiple save parts are separated.

Mapping to Abstract Syntax

| <basic state>(< *name* >, *triggers*, *)
=> **mk-***State-node*(*Mapping*(*name*),
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Identifier*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Input-node*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Continuous-signal*})
| <save part>(*s*) => *Mapping*(*s*)
The mapping to the abstract syntax is done selecting the appropriate parts for the AS1 grammar.

4.3.7.4 Input Node

Abstract Syntax

Input-node	::	*Signal-identifier* [*Variable-identifier*]* *Transition*

Conditions on Abstract Syntax

∀ in_1, in_2 ∈ *Input-node*: *parentAS1*(in_1)=*parentAS1*(in_2) ∧ $in_1 \neq in_2$ ⇒
s-*Signal-identifier*(in_1) ≠ **s-***Signal-identifier*(in_2)
The *Signal-identifier*s in the *Input-node*-**set** must be distinct.

∀ *in* ∈ *Input-node*: **let** *sd* = *in*.**s-***Signal-identifier*.*refersto1* **in**
length(**s-***Variable-identifier*-**seq**(*in*)) = *length*(**s-***Sort-name*-**seq**(*sd*)) ∧
∀*i* ∈ 1 .. *length*(**s-***Variable-identifier*-**seq**(*in*)):
let *vd* = *in*.**s-***Variable-identifier*-**seq**[*i*].*refersto1* **in**
sortCompatible(**s-***Sort-name*(*vd*), **s-***Sort-name*-**seq**(*sd*)[*i*])
endlet
endlet
The length of the list of optional *Variable-identifier*s must be the same as the number of *Sort-names* in the *Signal-definition* denoted by the *Signal-identifier* and the sorts of the variables must correspond by position to the sorts of the data items that can be carried by the signal.

Concrete Syntax

<input part> :: <input list> <transition>
<input list> = <stimulus>+
<stimulus> :: <signal list item> [[<variable>]+]

Conditions on Concrete Syntax

$\forall$ *s* $\in$ <stimulus>: *s*.**s-**<signal list item>*.refersto0* $\notin$ <remote variable definition>
A <signal list item> must not denote a <remote variable identifier>.

Transformations

< <input part>(< *s* > $^\frown$ *r*, *t*) > **provided** *r* $\neq$ *empty*
=3=> < <input part>(< *s* >, *t*), <input part>(*r*, *t*) >
Multiple inputs are separated.

Mapping to Abstract Syntax

| <input part>(< <stimulus>(*item*, *vars*) >, *trans*)
=> **mk-***Input-node*(*Mapping*(*item*),
if *vars* = *undefined* **then** *empty* **else** *Mapping*(*vars*) **endif**, *Mapping*(*trans*))

4.3.7.5 Continuous Signal

Abstract Syntax

Continuous-signal	::	*Continuous-expression Transition*
Boolean-expression	=	*Expression*
Continuous-expression	=	*Boolean-expression*

Conditions on Abstract Syntax

$\forall$ *c* $\in$ *Continuous-signal*:
sortCompatible(*exprSort*(*c*.**s-***Continuous-expression*), **mk-***Name*("Boolean"))
The continuous expression must be of type Boolean.

Concrete Syntax

<continuous signal> :: <continuous expression> <transition>
<continuous expression> = <Boolean<expression>

Mapping to Abstract Syntax

| <continuous signal>(*ex*, *trans*)
=> **mk-***Continuous-signal*(*Mapping*(*ex*), *Mapping*(*trans*))

4.3.7.6 Free Action

Abstract Syntax

Free-action	::	*Connector-name Transition*

Concrete Syntax

<label> :: <connector<name>
<free action> :: <transition> [<connector<name>]

Auxiliary Functions

getLabel(*t*: <transition>): <label> =$_{def}$
if *t*.**s-**<action statement> = *empty* **then** *t*.**s-**<terminator statement>.**s-**<label>
else *t*.**s-**<action statement>.*head*.**s-**<label>
endif
The function *getLabel* extracts the first label from the transition.

Conditions on Concrete Syntax

∀ *f* ∈ <free action>: *f*.**s-**<transition>.*getLabel* ≠ *undefined*
If the <transition string> of the <transition> in <free action> is non-empty, the first <action statement> must have a <label> otherwise the <terminator statement> must have a <label>.

Mapping to Abstract Syntax

| <free action>(*trans*, *)
=> **mk-***Free-action*(*Mapping*(*getLabel*(*trans*)), *Mapping*(*trans*))

4.3.8 Transition

4.3.8.1 Transition

Abstract Syntax

Transition :: *Graph-node** { *Terminator* | *Decision-node* }

Concrete Syntax

<transition> = <transition gen transition string> | <terminator statement>
<transition gen transition string> :: <transition string> [<terminator statement>]
<transition string> = {<action statement>}+

Conditions on Concrete Syntax

∀ *t* ∈ <transition gen transition string>:
t.**s-**<terminator statement> ≠ *undefined* ∨
t.parentAS0.parentAS0.parentAS0 ∈ <decision> ∨
(*t*.**s-**<transition string>.*last* ∈ <decision> ∧ *TerminatingDecision*(*t*.**s-**<transition string>.*last*))
If the <terminator> of a <transition> is omitted, then the last action in the <transition> must contain a terminating <decision>, except when a <transition> is contained in a <decision>.

Transformations

t=<terminator statement>(*,*) **provided** *t.parentAS0* ∉ <transition>
=2=> <transition gen transition string>(*empty*, *t*)
This rule unifies the two possible representations for <transition> into one. Please note, that the resulting structure would not be valid concrete syntax. However, this is remedied by the transformations for decisions, see Section 4.3.8.12.

Mapping to Abstract Syntax

| <transition gen transition string>(*s*, *t*)
=> **if** *t* = *undefined* **then mk-***Transition*(*Mapping*(< *x* **in** *s*: (*x* ∉ <decision>) >), *Mapping*(*s.last*))
else mk-*Transition*(*Mapping*(*s*), *Mapping*(*t*))
endif
The mapping to the abstract syntax must handle decisions specially. Please note that due to the transformations for decision there can be only one decision within one transition string and this decision is then the last element of the sequence. In this case there will be no terminating statement in the transition and the decision is mapped to the terminating statement of the AS1 transition.

4.3.8.2 Graph Node

Abstract Syntax

Graph-node	=	*Task-node*
	|	*Output-node*
	|	*Create-request-node*
	|	*Set-node*
	|	*Reset-node*

Concrete Syntax

<action statement> :: [<label>] <action 1>
<action 1> =
<task> | <output> | <create request> | <decision> | <set> | <reset> | <export> | <import>

Transformations

< *a*, <action statement>(*l*, *) > ⌢ *r* **provided** *l* ≠ *undefined*
=2=> < *a* >
and
a.parentAS0 => <transition gen transition string>(*a.parentAS0*.**s-**<action statement>,
<terminator statement>(*undefined*, <join>(*l*.**s-**<name>)))
and
let *p* = *parentAS0ofKind*(*a*,<free action> ∪ <state>) **in**
< *p* > => < *p*, <free action>(*r*, *undefined*) >

If a <label> is not the first label of a <transition string>, the <transition string> is split into two parts. All <action statements> preceding the <label> are preserved in the original transition, which is terminated with a <join> to the <label>. All action statements following <label> are copied to a new <free action>, which starts with the <label>.

Mapping to Abstract Syntax

| <action statement>(*, *x*) => *Mapping*(*x*)

4.3.8.3 Task

Abstract Syntax

Task-node	=	*Assignment*
Assignment	::	*Variable-identifier Expression*

Conditions on Abstract Syntax

∀ *a* ∈ *Assignment*: *a*.**s-***Variable-identifier.refersto1* ∈ *Variable-definition*

In an *Assignment*, the identifier must be a variable identifier.

∀ *a* ∈ *Assignment*: **let** *d* = *a*.**s-***Variable-identifier.refersto1* **in**
sortCompatible(*exprSort*(**s-***Expression*(*a*)), **s-***Sort-name*(*d*))
endlet

In an *Assignment*, the sort of the *Expression* must be equal to the sort of the *Variable-identifier*.

Concrete Syntax

<task> :: <textual task body>
<textual task body> = <assignment>
<assignment> :: <variable> <expression>
<variable> = <variable<identifier>

Mapping to Abstract Syntax

| <task>(<assignment>(*var*, *expr*)) => **mk-***Assignment*(*Mapping*(*var*), *Mapping*(*expr*))

4.3.8.4 Output

Abstract Syntax

Output-node	::	*Signal-identifier* [*Expression*]* [*Signal-destination*]
Signal-destination	=	*Expression*

Conditions on Abstract Syntax

∀ *n* ∈ *Output-node*: **let** *sd* = *in*.**s-***Signal-identifier.refersto1* **in**
length(**s-***Expression*-**seq**(*n*)) = *length*(**s-***Sort-name*-**seq**(*sd*)) ∧
∀*i* ∈ 1 .. *length*(**s-***Expression*-**seq**(*n*)):
sortCompatible(*exprSort*(**s-***Expression*-**seq**(*n*)[*i*]), **s-***Sort-name*-**seq**(*sd*)[*i*])
endlet

The length of the list of optional *Expression*s must be the same as the number of *Sort-name*s in the *Signal-definition* denoted by the *Signal-identifier*. Each *Expression* must be sort compatible to the corresponding (by position) *Sort-name* in the *Signal-definition.*

∀ *o* ∈ *Output-node*: *sortCompatible*(*exprSort*(*o*.**s-***Signal-destination*), **mk-***Name*("Pid"))

In an output, the destination must be of sort Pid.

∀ *o* ∈ *Output-node*: *o*.**s-***Signal-identifier.refersto1* ∈ *Signal-definition*

In an output, the identifier must be a signal identifier.

Concrete Syntax

<output> :: <output body>
<output body> :: <output body gen identifier>+ <communication constraints>
<output body gen identifier> :: <signal><identifier> [<actual parameters>]
<actual parameters> :: <actual parameter list>
<actual parameter list> =[<expression>]+
<communication constraints> = [<destination>]
<destination> = <expression>

Transformations

< <action statement>(*l*, <output>(<output body>(< *i* > ⌢ *r*, *C*))) > **provided** *r* ≠ *empty*
=3=> < <action statement>(*l*, <output>(<output body>(< *i* >, *C*))),
<action statement>(*undefined*, <output>(<output body>(*r*, *C*))) >

If several pairs of <signal identifier> and <actual parameters> are specified in an <output body>, this is derived syntax for specifying a sequence of <output>s in the same order as specified in the original <output body>, each containing a single pair of <signal identifier> and <actual parameters>. The **to** <destination> clause is repeated in each of the <output>s.

Mapping to Abstract Syntax

| <output>(<output body>(< <output body gen identifier>(*id*, *par*) >, *dest*))
=> **mk-***Output-node*(*Mapping*(*id*),
if *par*=*undefined* **then** *empty* **else** *Mapping*(*par*) **endif**, *Mapping*(*dest*))

4.3.8.5 Create

Abstract Syntax

Create-request-node	::	*Agent-identifier*

Conditions on Abstract Syntax

∀ *c* ∈ *Create-request-node*: *c*.**s-***Agent-identifier.refersto1* ∈ *Agent-definition*
In a create, the identifier must be an agent identifier.

Concrete Syntax

<create request> :: <create body>
<create body> = <identifier>

Mapping to Abstract Syntax

| <create request>(*id*) => **mk-***Create-request-node*(*Mapping*(*id*))

4.3.8.6 Set

Abstract Syntax

Set-node	::	*Time-expression Timer-identifier*
Time-expression	=	*Expression*

Conditions on Abstract Syntax

∀ *s* ∈ *Set-node*: *sortCompatible*(*exprSort*(*s*.**s-***Time-expression*), **mk-***Name*("Time"))
In a set node, the expression must be of sort Time.

Concrete Syntax

<set> :: <set clause>+
<set clause> :: <Time<expression> <timer<identifier>

Conditions on Concrete Syntax

∀ *s* ∈ <set clause>: *s*.**s-**<identifier>.*refersto0* ∈ <timer definition>
In a set clause, the identifier must be a timer identifier.

Transformations

< <action statement>(*l*, <set>(< *i* > ⌢ *r*)) > **provided** *r* ≠ *empty*
=3=> < <action statement>(*l*, <set>(< *i* >)), <action statement>(*undefined*, <set>(*r*)) >
A <set> may contain several <set clause>s. This is derived syntax for specifying a sequence of <set>s, one for each <set clause> such that the original order in which they were specified in <set> is retained.

Mapping to Abstract Syntax

| <set>(< <set clause>(*ex*, *id*) >) => **mk-***Set-node*(*Mapping*(*ex*), *Mapping*(*id*))

4.3.8.7 Reset

Abstract Syntax

Reset-node	::	*Timer-identifier*

Concrete Syntax

<reset> :: <reset clause>+
<reset clause> = <timer<identifier>

Conditions on Concrete Syntax

∀ *r* ∈ <reset>: ∀ *c* ∈ *r*.**s-**<reset clause>: *c*.**s-**<identifier>.*refersto0* ∈ <timer definition>
In a reset, the identifier must be a timer identifier.

Transformations

< <action statement>(*l*, <reset>(< *i* > ⌢ *r*)) > **provided** *r* ≠ *empty*
=3=> < <action statement>(*l*, <reset>(< *i* >)), <action statement>(*undefined*, <reset>(*r*)) >
A <reset> may contain several <reset clause>s. This is derived syntax for specifying a sequence of <reset>s, one for each <reset clause> such that the original order in which they were specified in <reset> is retained.

Mapping to Abstract Syntax

| <reset>(< *id* >) => **mk-***Reset-node*(*Mapping*(*id*))

4.3.8.8 Terminator

Abstract Syntax

Terminator	=	*Nextstate-node*
	|	*Stop-node*
	|	*Join-node*

Concrete Syntax

<terminator statement> :: [<label>] <terminator 2>
<terminator 2> =<nextstate> | <join> | <stop>

Mapping to Abstract Syntax

| <terminator statement>(*, *t*) => *Mapping*(*t*)

4.3.8.9 Nextstate

Abstract Syntax

Nextstate-node	::	*State-name*

Conditions on Abstract Syntax

∀*nn* ∈ *Nextstate-node*: *nn*.**s-***State-name.referstoName1* ≠ *undefined*
The *State-name* specified in a nextstate must be the name of a state within the same *State-transition-graph*.

Concrete Syntax

<nextstate> :: <nextstate body>
<nextstate body> = <state<name>

Mapping to Abstract Syntax

| <nextstate>(*n*) => **mk-***Nextstate-node*(*Mapping*(*n*))

4.3.8.10 Stop

Abstract Syntax

Stop-node	::	()

Concrete Syntax

<stop>= **stop**

Mapping to Abstract Syntax

| **stop** => **mk-***Stop-node*()

4.3.8.11 Join

Abstract Syntax

Join-node	::	*Connector-name*

Conditions on Abstract Syntax

∀*jn* ∈ *Join-node*: *jn*.**s-***Connector-name.referstoName1* ≠ *undefined*
The *Connector-name* in a join must be the name of a free action within the same *State-transition-graph*.

Concrete Syntax

<join> :: <connector<name>

Mapping to Abstract Syntax

| <join>(*n*) => **mk-***Join-node*(*Mapping*(*n*))

4.3.8.12 Decision

Abstract Syntax

Decision-node	::	*Decision-question* *Decision-answer*-**set** [*Else-answer*]
Decision-question	=	*Expression*
Decision-answer	::	*Constant-expression*-**set** *Transition*
Else-answer	::	*Transition*

Conditions on Abstract Syntax

∀ *da* ∈ *Decision-answer*: ∀ *ce* ∈ *da*.**s-***Constant-expression*-**set**:
sortCompatible(*exprSort*(*ce*), *exprSort*(**s-***Decision-question*(*da.parentAS1*)))
The *Constant-expression*s of the *Decision-answer*s must be sort compatible to the sort of the *Decision-question*.

∀ da_1 ∈ *Decision-answer*: ∀ da_2 ∈ *Decision-answer*:
$da_1 \neq da_2 \wedge parentAS1(da_1) = parentAS1(da_2) \Rightarrow$
{ *evalExpr*(*e*) | *e* ∈ da_1.**s-***Constant-expression* } ∩
{ *evalExpr*(*e*) | *e* ∈ da_2.**s-***Constant-expression* } = ∅
The *Constant-expression*s of the *Decision-answer*s must be mutually exclusive.

Concrete Syntax

<decision> :: <question> <decision body>
<decision body> :: <answer part>+ [<else part>]
<answer part> :: <answer> [<transition>]
<answer> = <constant expression>+
<else part> :: [<transition>]
<question> = <expression>

Auxiliary Functions

findContinueLabel(*x*: *DefinitionAS0*): <name> =$_{def}$
if *x* ∈ <transition gen transition string> ∧ *x*.**s-**<terminator statement> ≠ *undefined* ∧
x.**s-**<terminator statement>.**s-**<label> = *undefined* ∧
x.**s-**<terminator statement>.**s-**<terminator 2> ∈ <join>
then *x*.**s-**<terminator statement>.**s-**<terminator 2>.**s-**<name>
else *findContinueLabel*(*x.parentAS0*)
endif

The function *findContinueLabel* computes the continuation label after a decision within a transition string.

TerminatingDecision(*d*: <decision>): *BOOLEAN* =$_{def}$
(∀ *a* ∈ *d*.**s-**<answer part>: *TerminatingTransition*(*a*.**s-**<transition>)) ∧
(*d*.**s-**<else part> = *undefined* ∨ *TerminatingTransition*(*d*.**s-**<else part>.**s-**<transition>))

A <decision> is a terminating decision, if each <answer part> and <else part> in its <decision body> is a terminating <answer part> or <else part> respectively.

TerminatingTransition(*t*: <transition>): *BOOLEAN* =$_{def}$
t ∈ <terminator statement> ∨
t.**s-**<terminator statement> ≠ *undefined* ∨
(**let** *d*= *t*.**s-**<action statement>.*last* **in** *d* ∈ <decision> ∧ *TerminatingDecision*(*d*) **endlet**)

An <answer part> or <else part> in a decision is a terminating <answer part> or <else part> respectively if it contains a <transition> where a <terminator statement> is specified, or contains a <transition string> whose last <action statement> contains a terminating decision.

Transformations

<else part>(*undefined*) =2=> <else part>(<transition gen transition string>(*empty*, *undefined*))
<answer part>(*a*, *undefined*) =2=>
<answer part>(*a*, <transition gen transition string>(*empty*, *undefined*))

These first two transformations are used to insert an empty transition instead of an undefined one. This empty transition will be filled with a terminator within the step below (inserting terminating actions into the transition).

t=<transition gen transition string>(*a*, *undefined*)
provided *a.last* ∉ <decision> ∧ *t.parentAS0.parentAS0.parentAS0* ∈ <decision> ∧
t.findContinueLabel ≠ *undefined*
=2=> <transition gen transition string>(*a*,
<terminator statement>(*undefined*,<join>(*findContinueLabel*(*t*))))

If a <decision> is not terminating then it is derived syntax for a <decision> wherein all not terminating <answer part>s and the <else part> if not terminating have inserted at the end of their <transition> a <join> to the first <action statement> following the decision or if the decision is the last <action statement> in a <transition string> to the following <terminator statement>.

< *d* =<decision>(*, *), <action statement>(*undefined*, *a*) > **provided** ¬ *TerminatingDecision*(*d*)
=2=> < *d*, <action statement>(*newName*(*undefined*), *a*) >

<transition gen transition string>(*str*, <terminator statement>(*undefined*, *t*))
provided *str.last* ∈ <decision> ∧ ¬ *str.last.TerminatingDecision*
=2=> <transition gen transition string>(*str*, <terminator statement>(*newName*(*undefined*), *t*))

The rules above insert a new label after a non-terminating decision.

Mapping to Abstract Syntax

| <decision>(*ex*, <decision body>(*ans*, *els*))
=> **mk-***Decision-node*(*Mapping*(*ex*), *Mapping*(*ans*).*toSet*, *Mapping*(*els*))

4.3.9 Expression

4.3.9.1 Expression

Abstract Syntax

Expression	=	*Constant-expression*
	|	*Active-expression*
Constant-expression	=	*Literal*
	|	*Operation-application*
Active-expression	=	*Variable-access*
	|	*Operation-application*
	|	*Imperative-expression*
Imperative-expression	=	*Now-expression*
	|	*Pid-expression*
	|	*Timer-active-expression*

Please note, that the above definition could be simplified. This can be done by omitting the difference between active expressions and constant expressions. This difference does not show up at any place, so it could be simply dropped. It is provided here as is to preserve the similarity to the SDL definition.

Concrete Syntax

<constant expression> = <constant<expression>
<expression> = <operand>

<operand> = <operand0> | <operand gen operand>
<operand gen operand> :: <operand> <operand0> /* => */

<operand0> =<operand1> | <operand0 gen operand0>
<operand0 gen operand0> :: <operand0> {**or** | **xor**} <operand1>

<operand1> =<operand2> | <operand1 gen operand1>
<operand1 gen operand1> :: <operand1> <operand2> /* **and** */

<operand2> =<operand3> | <operand2 gen operand2>
<operand2 gen operand2> :: <operand2>
{ <greater than sign> | <greater than or equals sign> | <less than sign>
| <less than or equals sign> | <equals sign> | <not equals sign> }
<operand3>

<operand3> =<operand4> | <operand3 gen operand3>
<operand3 gen operand3> :: <operand3> { <plus sign> | <hyphen> } <operand4>

<operand4> =<operand5> | <operand4 gen operand4>
<operand4 gen operand4> ::
<operand4> { <asterisk> | <solidus> | **mod** } <operand5>

<operand5> ::[<hyphen> | **not**] <primary>

<primary> =
<operation application>
/* | <literal> this is disabled because it is subsumed by <variable access> */
| <primary gen expression>
| <active primary>
<primary gen expression> :: <expression>

<active primary> = <variable access> | <imperative expression>
<imperative expression> =
<now expression> | <pid expression> | <timer active expression>

Transformations

<operand gen operand>(*x*, *y*) =3=> <operation application>("=>", <*x*, *y*>)
<operand0 gen operand0>(*x*, **or**, *y*) =3=> <operation application>("or", <*x*, *y*>)
<operand0 gen operand0>(*x*, **xor**, *y*) =3=> <operation application>("xor", <*x*, *y*>)
<operand1 gen operand1>(*x*, *y*) =3=> <operation application>("and", <*x*, *y*>)
<operand2 gen operand2>(*x*,<greater than sign>, *y*) =3=> <operation application>(">", <*x*, *y*>)
<operand2 gen operand2>(*x*,<greater than or equals sign>, *y*)
=3=> <operation application>(">=", <*x*, *y*>)
<operand2 gen operand2>(*x*,<less than sign>, *y*) =3=> <operation application>("<", <*x*, *y*>)
<operand2 gen operand2>(*x*,<less than or equals sign>, *y*)
=3=> <operation application>("<=", <*x*, *y*>)
<operand2 gen operand2>(*x*,<equals sign>, *y*) =3=> <operation application>("=", <*x*, *y*>)
<operand2 gen operand2>(*x*,<not equals sign>, *y*) =3=> <operation application>("/=", <*x*, *y*>)
<operand3 gen operand3>(*x*,<plus sign>, *y*) =3=> <operation application>("+", <*x*, *y*>)
<operand3 gen operand3>(*x*,<hyphen>, *y*) =3=> <operation application>("-", <*x*, *y*>)
<operand4 gen operand4>(*x*,<asterisk>, *y*) =3=> <operation application>("*", <*x*, *y*>)
<operand4 gen operand4>(*x*,<solidus>, *y*) =3=> <operation application>("/", <*x*, *y*>)
<operand4 gen operand4>(*x*, **mod**, *y*) =3=> <operation application>("mod", <*x*, *y*>)
<operand5>(<hyphen>, *x*) =3=> <operation application>("-", <*x*>)
<operand5>(**not**, *x*) =3=> <operation application>("not", <*x*>)

An expression of the form
<expression> <infix operation name> <expression>
is derived syntax for
<infix operation name> **(** <expression>**,** <expression> **)**
where <infix operation name> represents an *Operation-name* although the name would not be valid concrete syntax.
Similarly,
<monadic operation name> <expression>
is derived syntax for
<monadic operation name> **(** <expression> **)**
where <monadic operation name> represents an *Operation-name*.

Mapping to Abstract Syntax

| <primary gen expression>(*x*) => *Mapping*(*x*)
| <operand5>(*undefined*, *x*) => *Mapping*(*x*)

The mappings above are used for those expression constructors that do not get transformed away. After the transformation only the two constructors above remain.

4.3.9.2 Literal

Abstract Syntax

Literal :: *Literal-name*

Conditions on Abstract Syntax

∀*l* ∈ *Literal*: *l*.**s-***Literal-name*.**s-***TOKEN.isPredefLiteral*

Any literal must be a predefined literal.

Concrete Syntax

The concrete syntax of <literal> is subsumed within <variable access> (Section 4.3.9.4) and will be distinguished in the mapping of <identifier> (Section 4.3.2.2).

4.3.9.3 Operation Application

Abstract Syntax

Operation-application :: *Operation-name Expression+*

Conditions on Abstract Syntax

∀ *o* ∈ *Operation-application*: *isPredefOperation*(*o*.**s-***Name*.**s-***TOKEN*) ∧
< *e.exprSort*.**s-***TOKEN* | *e* **in** *o*.**s-***Expression*-**seq** > ∈ *correctTypes*(*o*.**s-***Name*.**s-***TOKEN*)

The operation application must refer to a predefined operator and the argument types must be correct.

Concrete Syntax

<operation application> :: <operator<name> <expression list>
<expression list> = <expression>+

Mapping to Abstract Syntax

| <operation application>(*name*, *params*) =>
mk-*Operation-application*(*Mapping*(*name*), *Mapping*(*params*))

4.3.9.4 Variable Access

Abstract Syntax

Variable-access	=	*Variable-identifier*

Concrete Syntax

<variable access> = <variable<identifier>

Conditions on Concrete Syntax

∀ *v* ∈ <identifier>: *v.parentAS0* ∈ <operand5> ⇒ *v*.**s-**<identifier>.*refersto0* ∈ <variable definition>

A variable access must refer to a variable.

Mapping to Abstract Syntax
This is already defined with the mapping of <identifier>.

4.3.9.5 Now Expression

Abstract Syntax

Now-expression	::	()

Concrete Syntax

<now expression> = **now**

Mapping to Abstract Syntax

| **now** => **mk-***Now-expression*()

4.3.9.6 Pid Expression

Abstract Syntax

Pid-expression	=	*Self-expression*
	|	*Parent-expression*
	|	*Offspring-expression*
	|	*Sender-expression*
Self-expression	::	()
Parent-expression	::	()
Offspring-expression	::	()
Sender-expression	::	()

Concrete Syntax

<pid expression> =
self
| **parent**
| **offspring**
| **sender**

Mapping to Abstract Syntax

| **self** => **mk-***Self-expression*()
| **parent** => **mk-***Parent-expression*()
| **offspring** => **mk-***Offspring-expression*()
| **sender** => **mk-***Sender-expression*()

4.3.9.7 Timer Active Expression

Abstract Syntax

Timer-active-expression :: *Timer-identifier*

Concrete Syntax

<timer active expression> :: <timer<identifier>

Conditions on Concrete Syntax

$\forall$ *t* $\in$ <timer active expression>: *t*.**s-**<identifier>.*refersto0* $\in$ <timer definition>
An identifier in a timer active expression must refer to a timer definition.

Mapping to Abstract Syntax

| <timer active expression>(*id*) => **mk-***Timer-active-expression*(*Mapping*(*id*))

This following closing keyword of the case expression completes the definition of the mapping function.

endcase

4.4 Dynamic Semantics Overview

As already said within Section 1.4.2, the dynamic semantics is defined based on abstract state machines (ASM). Starting from there, a kind of a package for RSDL is created which is called SAM. This package includes behaviour primitives for RSDL as well as basic connection agents that correspond to RSDL channels. Moreover, the representation of structural properties is possible within SAM.
As the next step after the static analysis (condition checks and transformations), the abstract syntax structure is mapped to the SAM framework. The structural AS information is mapped via initialisation to the SAM structures and connections. The behaviour parts of the AS are mapped to the behaviour primitives of SAM. There is one more part in the abstract syntax, namely the data. This is factored out using an interface. With this interface it is possible to define the dynamics of the language without knowing the specialities of the data part and also to define the data semantics without knowing the dynamic part. See also Section 4.4.4 below.

4.4.1 Special Abstract Machine - SAM

The definition of SAM comprises a general part of definitions for RSDL and a special part of the definitions for the behaviour primitives. In the general part the RSDL agents are defined and their interfaces defined using gates. Also the connections are defined as a means to capture channels.

4.4.2 Initialisation

The initialisation involves the creation of the system structure as defined by the RSDL specification. This is done in a straightforward way as defined by the RSDL semantics. The starting point is a system agent that refers to the system definition. This will include internal block agents and channels. These are also all initialised together

with the gates that are used for the connections. The whole process stops when it comes to the behaviour of the agents which is captured by the compilation. The initialisation does only state that the agents will act on the compiled behaviour parts. How they process the result of the compilation is already stated within the SAM.

4.4.3 Compilation

The compilation process defines how the special RSDL behaviour is mapped to the behaviour structure as defined by the SAM. As the SAM behaviour features are designed to match those of RSDL this transformation is relatively simple (almost one-to-one).

4.4.4 Data

The data semantics is given with the implementation of the interface functions for the data interface. These functions do capture only the static aspects of the data semantics, all dynamic aspects are given with the dynamic semantics part.
The semantics of the data type part of RSDL will be handled separately from the concurrency related aspects of the language. To make this splitting possible, an interface for the semantics definition has to be defined. Because the RSDL data part is very simple, the implementation of the interface functions is given together with their declaration in this book.
We start the interface with a domain for values, called *VALUE* and a domain for variable names. The value domain is defined within the data part whereas the variable names are defined within the concurrency part.

$$VALUE =_{def} RSDLINTEGER \cup RSDLBOOLEAN \cup RSDLTIME \cup RSDLDURATION \cup RSDLAGENT$$

$$VARIABLENAME =_{def} Identifier$$

An important part of the data interface is a domain *STATE*. This domain is abstract from the concurrency side, and concrete from the data type side. It represents the values of the variables of an agent. This is achieved by a dynamic, controlled function *state* defined on process instances.

$$STATE =_{def} VARIABLENAME \rightarrow VALUE$$

controlled *state*: $AGENT \rightarrow STATE$

The *state* function will be changed dynamically whenever the state of an RSDL agent changes. It is solely used within the concurrency semantics part. The data type semantics part provides the initial value for this function via the function *initAgentState*.

$$DECLARATIONS =_{def} Variable\text{-}definition\textbf{-set}$$

$$initAgentState: DECLARATIONS \rightarrow STATE$$
$$initAgentState(d: DECLARATIONS): STATE =_{def}$$
$$\{ \{v.myFullIdentifierAS1 \rightarrow v.\textbf{s-}Constant\text{-}expression.evalExpr\} \mid v \in d \}$$

The data type part has to provide a function how assignments are performed, namely

$$assign: VARIABLENAME \times VALUE \times STATE \rightarrow STATE$$
$$assign(vari: VARIABLENAME, val: VALUE, stat: STATE): STATE =_{def}$$
$$stat \setminus \{ vari \rightarrow stat(vari) \} \cup \{ vari \rightarrow val \}$$

There is a rule macro using this function, which is doing the real assignment.

$$\text{ASSIGN}(variableName: VARIABLENAME, value: VALUE, agent: AGENT) \equiv$$
$$agent.state := assign(variableName, value, agent.state)$$

Please note, that assignments are the only way to change the state.
In order to get the current value of a variable, the data part provides a function to get it.

$$eval: VARIABLENAME \times STATE \rightarrow VALUE$$
$$eval(v: VARIABLENAME, s: STATE): VALUE =_{def} s(v)$$

Finally, there are various functions to model the predefined functions. There is one function computing the result in this case.

compute: *Name* × *VALUE** → *VALUE*

compute(*n*: *Name*, *args*: *VALUE**): *VALUE* =$_{\text{def}}$
if *n*.**s-***TOKEN* = ”true” **then selector-***RSDLBOOLEAN*(*True*)
elseif *n*.**s-***TOKEN* = ”false” **then selector-***RSDLBOOLEAN*(*False*)
elseif *n*.**s-***TOKEN* = ”null” **then selector-***RSDLAGENT*(*nullAgent*)
elseif *n*.**s-***TOKEN.isIntToken* **then selector-***RSDLINTEGER*(*n*.**s-***TOKEN.getIntValue*)
elseif *n*.**s-***TOKEN.isRealToken* **then selector-***RSDLDURATION*(*n*.**s-***TOKEN.getRealValue*)
elseif *n*.**s-***TOKEN* = ”=” **then selector-***RSDLBOOLEAN*(*args.head* = *args.tail.head*)
elseif *n*.**s-***TOKEN* = ”/=” **then selector-***RSDLBOOLEAN*(*args.head* ≠ *args.tail.head*)
elseif *n*.**s-***TOKEN* = ”<=”
then selector-*RSDLBOOLEAN*(*args.head*.**selectKind-***RSDLINTEGER* ≤
args.tail.head.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”>=”
then selector-*RSDLBOOLEAN*(*args.head*.**selectKind-***RSDLINTEGER* ≥
args.tail.head.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”<”
then selector-*RSDLBOOLEAN*(*args.head*.**selectKind-***RSDLINTEGER* <
args.tail.head.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”>”
then selector-*RSDLBOOLEAN*(*args.head*.**selectKind-***RSDLINTEGER* >
args.tail.head.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”+” ∧ *args.head* ∈ *RSDLINTEGER*
then selector-*RSDLINTEGER*(*args.head*.**selectKind-***RSDLINTEGER* +
args.tail.head.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”+” ∧ *args.head* ∈ *RSDLTIME*
then selector-*RSDLTIME*(*args.head*.**selectKind-***RSDLTIME* +
args.tail.head.**selectKind-***RSDLDURATION*)
elseif *n*.**s-***TOKEN* = ”-” ∧ *args.tail* = *empty*
then selector-*RSDLINTEGER*(- *args.head*.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”-” ∧ *args.tail* ≠ *empty* ∧ *args.head* ∈ *RSDLINTEGER*
then selector-*RSDLINTEGER*(*args.head*.**selectKind-***RSDLINTEGER* -
args.tail.head.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”-” ∧ *args.tail* ≠ *empty* ∧ *args.head* ∈ *RSDLTIME*
then selector-*RSDLTIME*(*args.head*.**selectKind-***RSDLTIME* -
args.tail.head.**selectKind-***RSDLDURATION*)
elseif *n*.**s-***TOKEN* = ”*”
then selector-*RSDLINTEGER*(*args.head*.**selectKind-***RSDLINTEGER* *
args.tail.head.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”/”
then selector-*RSDLINTEGER*(*args.head*.**selectKind-***RSDLINTEGER* /
args.tail.head.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”mod”
then selector-*RSDLINTEGER*(*args.head*.**selectKind-***RSDLINTEGER* mod
args.tail.head.**selectKind-***RSDLINTEGER*)
elseif *n*.**s-***TOKEN* = ”and”
then selector-*RSDLBOOLEAN*(*args.head*.**selectKind-***RSDLBOOLEAN* ∧
args.tail.head.**selectKind-***RSDLBOOLEAN*)
elseif *n*.**s-***TOKEN* = ”or”
then selector-*RSDLBOOLEAN*(*args.head*.**selectKind-***RSDLBOOLEAN* ∨
args.tail.head.**selectKind-***RSDLBOOLEAN*)
elseif *n*.**s-***TOKEN* = ”=>”
then selector-*RSDLBOOLEAN*(*args.head*.**selectKind-***RSDLBOOLEAN* ⇒
args.tail.head.**selectKind-***RSDLBOOLEAN*)
elseif *n*.**s-***TOKEN* = ”not” **then selector-***RSDLBOOLEAN*(¬ *args.head*.**selectKind-***RSDLBOOLEAN*)
else *undefined*
endif

The next function is used to evaluate constant expressions.

evalExpr(*e*: *Expression*): *VALUE* =$_{\text{def}}$
if *e* ∈ *Literal* **then** *compute*(*e*.**s-***Name*, *empty*)
elseif *e* ∈ *Operation-application* **then**
compute(*e*.**s-***Name*, < *evalExpr*(*arg*) | *arg* **in** *e*.**s-***Expression*-**seq** >)
else *undefined*
endif

Now we introduce various predefined functions to handle predefined literals. They are not defined here because they have a standard meaning.

isIntToken: *TOKEN* → *BOOLEAN*
isRealToken: *TOKEN* → *BOOLEAN*
getIntValue: *TOKEN* → *INT*
getRealValue: *TOKEN* → *REAL*

The next functions are used to formalise the signatures of predefined literals and operations.

predefSignature(*t*: *TOKEN*): (*TOKEN* × *TOKEN**)-**set** =$_{\text{def}}$
if *t* ∈ { "<", ">", "<=", ">=" } **then** { < "Boolean", < "Integer", "Integer" > > }
elseif *t* ∈ { "*", "/", "mod" } **then** { < "Integer", < "Integer", "Integer" > > }
elseif *t* = "-" **then** { < "Integer", < "Integer" > >, < "Integer", < "Integer", "Integer" > >,
< "Time", < "Time", "Duration" > > }
elseif *t* = "+" **then**
{ < "Integer", < "Integer", "Integer" > >, < "Time", < "Time", "Duration" > > }
elseif *t* ∈ { "and", "or" } **then** { < "Boolean", < "Boolean", "Boolean" > > }
elseif *t* ="not" **then** { < "Boolean", < "Boolean" > > }
elseif *t* ∈ { "=", "/=" } **then**
{ < "Boolean", < *x*, *x* > > | *x* ∈ { "Boolean", "Integer", "Time", "Duration", "PId" } }
else ∅
endif

literalSort(*t*: *TOKEN*): *TOKEN* =$_{\text{def}}$
if *t* ∈ { "true", "false" } **then** "Boolean"
elseif *t* = "none" **then** "PId"
elseif *t*.*isIntToken* **then** "Integer"
elseif *t*.*isRealToken* **then** "Duration"
else *undefined*
endif

isPredefLiteral(*t*: *TOKEN*): *BOOLEAN* =$_{\text{def}}$ *t.literalSort* ≠ *undefined*

isPredefOperation(*t*: *TOKEN*): *BOOLEAN* =$_{\text{def}}$ *t.predefSignature* ≠ ∅

correctTypes(*t*: *TOKEN*, *args*: *TOKEN**): *BOOLEAN* =$_{\text{def}}$ { *sig.last* | *sig* ∈ *t.predefSignature* }

operationSort(*t*: *TOKEN*, *args*: *TOKEN**): *BOOLEAN* =$_{\text{def}}$
let *candidates* = { *sig.head* | *sig* ∈ *t.predefSignature* : *sig.last* = *args* } **in** *candidates.take* **endlet**

Moreover, the following domains and functions referring to the predefined data are used.

RSDLTIME =$_{\text{def}}$ *REAL*
RSDLBOOLEAN =$_{\text{def}}$ *BOOLEAN*
RSDLINTEGER =$_{\text{def}}$ *INT*
RSDLDURATION =$_{\text{def}}$ *REAL*
RSDLPID =$_{\text{def}}$ *AGENT*

We also need a predefined *null* agent.

static *nullAgent*: → *AGENT*

Finally, we introduce various conversion functions.

value2bool(*b*: *VALUE*): *BOOLEAN* $=_{def}$ **selectKind-***RSDLBOOLEAN*(*b*)
bool2value(*b*: *BOOLEAN*): *VALUE* $=_{def}$ **selector-***RSDLBOOLEAN*(*b*)
value2agent(*a*: *VALUE*): *AGENT* $=_{def}$ **selectKind-***RSDLAGENT*(*a*)
agent2value(*a*: *AGENT*): *VALUE* $=_{def}$ **selector-***RSDLAGENT*(*a*)
value2time(*a*: *VALUE*): *RSDLTIME* $=_{def}$ **selectKind-***RSDLTIME*(*a*)

This is the complete interface between the data part and the dynamic semantics together with the definition of the functions. The following special point is worth noting. The values for the predefined variables of an agent (**SENDER**, **PARENT**, **OFFSPRING**, **SELF**), as well as the value of **NOW** are provided by the concurrency part.
The following table lists again all the interface domains and functions as provided by the two parts.

Defined within the concurrency part	Defined within the data part
VARIABLENAME	*VALUE*
nullAgent	*RSDLBOOLEAN*, *RSDLINTEGER*, *RSDLTIME*, *RSDLDURATION*, *RSDLPID*
state	*STATE*
DECLARATIONS	*initAgentState*
ASSIGN	*assign*
	eval, *compute*, *evalExpr*
	isPredefLiteral, *literalSort*
	isPredefOperation, *correctTypes*, *operationSort*
	value2bool, *bool2value*, *value2agent*, *agent2valuevalue2time*

4.5 Special Abstract Machine Definition

The definition of the RSDL Abstract Machine is structured as follows:
- Signal Flow Related Definitions: Section 4.5.1,
- RSDL Agent Related Definitions: Section 4.5.2,
- Signal Processing Related Definitions: Section 4.5.2.3, and
- Behaviour Primitives: Section 4.5.4.

4.5.1 Signal Flow Model

This section introduces the signal flow model as part of the SAM. The main focus here is on a uniform treatment of signal flow aspects, in particular, on defining how *active objects* communicate through signals via *gates*. Also timers, which are modelled as special kinds of signals, are covered here.

4.5.1.1 Signals

PLAINSIGNAL represents the set of *signal types* as declared by an RSDL specification.

PLAINSIGNAL $=_{def}$ *Signal-definition*

SIGNAL $=_{def}$ *DefinitionAS1*

In an RSDL specification, also timers behave like signals, so we introduce a common domain *SIGNAL* for all of them. Dynamically created *signal instances* (*signals* for short) are introduced as elements of a dynamic domain *SIGNALINST*. Each element of *SIGNALINST* is uniquely related to an element of *SIGNAL*.

shared domain *PLAINSIGNALINST*

SIGNALINST $=_{def}$ *PLAINSIGNALINST* ∪ *TIMERINST*

For each signal the sender of the signal and the receiver constraint are stored in the functions *sigSender* and *toArg*, respectively. For each of the functions introduced in the following, we distinguish the general variant on

*SIGNALINST*s from the plain one on *PLAINSIGNALINST*. The complete definition of the derived version on *SIGNALINST*s can be found in Section 4.5.1.5.

signalType: *SIGNALINST* → *SIGNAL*
sigSender: *SIGNALINST* → *AGENT*
toArg: *SIGNALINST* → *TOARG*

TOARG =$_{def}$ *RSDLPID*

shared *plainSignalType*: *PLAINSIGNALINST* → *SIGNAL*
shared *plainSigSender*: *PLAINSIGNALINST* → *AGENT*
shared *plainToArg*: *PLAINSIGNALINST* → *TOARG*

With each signal a (possibly empty) list of *signal values* is associated. Since the type information and concrete value for signal values is immaterial to the dynamic aspects considered here, values are abstractly represented in a uniform way as elements of the domain *VALUE* as provided by the data interface in Section 4.4.4.

values: *SIGNALINST* → *VALUE**
shared *plainValues*: *PLAINSIGNALINST* → *VALUE**

4.5.1.2 Gates

Exchange of signals between RSDL block agents and with the environment is modelled by means of *gates* from a controlled domain *GATE*.

controlled domain *GATE*

A gate forms an interface for *serial* and *unidirectional* communication between two or more agents. Accordingly, gates are either classified as *input gates* or *output gates*.

DIRECTION =$_{def}$ {*inDir*, *outDir*}

controlled *direction*: *GATE* → *DIRECTION*

Discrete Delay Model
Signals need not reach their destination instantaneously but may be subject to delays. That means, it must be possible to send signals to arrive in the future. Although those signals are not available at their destination before their arrival time has come, they are to be associated with their destination gates. A gate must be capable of holding signals that are in transit. Hence, to each gate a *signal queue* is assigned, as detailed below.
To model signal arrivals at specified destination gates, each signal instance *s* which is currently in transit has an individual arrival time *s.arrival*, *s.arrival* > *now*, determining the time at which *s* eventually reaches a certain gate.

shared *arrival*: *SIGNALINST* → *REAL*

One can now represent the relation between signals and gates in a given SAM state by means of a dynamic function *schedule* defined on gates,

shared *schedule*: *GATE* → *SIGNALINST**

where *schedule* specifies for every gate *g* in *GATE* the corresponding *signal arrivals* at *g*.
An integrity constraint on *g.schedule* is that signals in *g.schedule* are linearly ordered by their arrival times. That is, if *g.schedule* contains signals *s*, *s',* and *s.arrival* < *s'.arrival*, then *s* < *s'* in the order as imposed by *g.schedule*. This condition is assured by the *insert* function below.

Waiting Signals
A signal instance *s* in *g.schedule* does not arrive "physically" at gate *g* before *now* ≥ *s.arrival*. Intuitively, that means that *s* remains "invisible" at *g* as long as it is in transit. Thus, in every given SAM state, the visible part of *g.schedule* forms a possibly empty signal queue,

queue: *GATE* → *SIGNALINST**

where *g.queue* represents those signal instances *s* in *g.schedule* which have already arrived at *g* but are still waiting to be removed from *g.schedule*. The visible part of *g* is denoted as *g.queue* and formally defined as follows. See also Figure 10 below for an overview of the functions on schedules.

queue(*g*: *GATE*): *SIGNALINST** =$_{\text{def}}$ < *s* **in** *g.schedule*: (*now* ≥ *s.arrival*) >

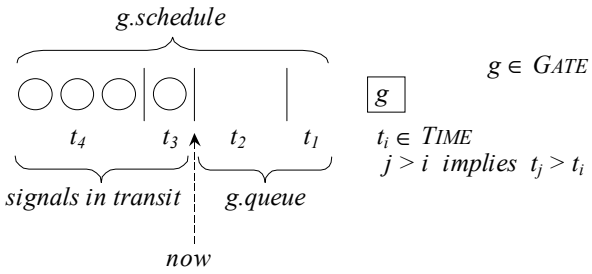


Figure 10: Signal instances at a gate

Operations on Schedules
To ensure that the order on signals is preserved when new signals are added to the schedule of a gate, there is a special insertion function on schedules. The result of inserting some signal *s* with the intended arrival time *t* into a finite signal list *seq* as represented by the schedule of a given gate is defined as follows.

insert(*s*: *SIGNALINST*, *t*: *RSDLTIME*, *seq*: *SIGNALINST**): *SIGNALINST** =$_{\text{def}}$
if *seq* = *empty* ∨ *t* < *seq.head.arrival* **then** < *s* > ^ *seq*
else < *seq.head* > ^ *insert*(*s*, *t*, *seq.tail*)

Analogously, a function *delete* is used to remove an item from a given schedule *seq*.

delete(*s*: *SIGNALINST*, *seq*: *SIGNALINST**): *SIGNALINST** =$_{\text{def}}$
if *seq* = *empty* **then** *empty*
elseif *seq.head* = *s* **then** *seq.tail*
else < *seq.head* > ^ *delete*(*s*, *seq.tail*)

As shorthand notations for operations used to update the schedule of a gate *g* (by assigning a new signal list to *g.schedule*) the following two rule macros are defined here. These macros will be used in subsequent rules.

INSERT(*s*: *SIGNALINST*, *t*: *REAL*, *g*: *GATE*) ≡
g.schedule := *insert*(*s*,*t*,*g.schedule*)
s.arrival := *t*

DELETE(*s*: *SIGNALINST*, *g*: *GATE*) ≡
g.schedule := *delete*(*s*,*g.schedule*)
s.arrival := *undefined*

4.5.1.3 Channels

Channel paths, as declared in a given RSDL specification, are abstractly represented as elements of a domain *CHANNEL*. An RSDL channel consists of one or two unidirectional *channel paths,* each of which is represented as an object of a dynamic domain *LINK*. Channel paths are associated to *LINK*s as stated by means of a corresponding function *channel*.

CHANNEL =$_{\text{def}}$ *Channel-path*
LINK =$_{\text{def}}$ *AGENT*
controlled *channel*: *LINK* → *CHANNEL*

Intuitively, elements of *LINK* are point-to-point connection primitives for the transport of signals. Each *l* of *LINK* is able to convey the signal types specified by *l.with*, from an originating gate *l.from* to a destination gate *l.to*.

controlled *from*: *LINK* → *GATE*
controlled *to*: *LINK* → *GATE*

with: *LINK* → *SIGNAL*-**set**
with(*l*: *LINK*): *SIGNAL*-**set** =$_{\text{def}}$ { *s.refersto1* | *s* ∈ *l.channel*.**s-***Signal-identifier*-**set** }

Signal Delays
RSDL channels are reliable and order-preserving communication links. A channel may however delay the transport of a signal for an *indeterminate* and *non-constant* time interval. Although the exact delaying behaviour is not further specified, the fact that channels are reliable implies that all delays must be finite.
Signal delays are modelled through a monitored function *delayedTime* stating the dependency on external conditions and events. In a given SAM state, *delayedTime* associates finite time intervals from the domain *REAL* to the elements of *LINK*, where the duration of a particular signal delay appears to be chosen non-deterministically.

monitored *delayedTime*: *LINK* → *REAL*

There are three integrity constraints on the function *delayedTime*:
1. For every link agent *l*, *l.delayedTime* ≥ *now*.
2. For every link to the input gate of an agent *l*, *l.delayedTime* = *now* (i.e. they are non-delaying).
3. For every link agent *l* the values of *l.delayedTime* increase monotonically (with respect to *now*).

Notice that the third integrity constraint is needed in order to ensure that channel paths are *order-preserving*, i.e. signals which are transported via the same channel path (and therefore are inserted into the same destination schedule) cannot overtake.

Channel Behaviour
A link agent *l* performs a single operation: signals received at gate *l.from* are forwarded to gate *l.to*. That means, *l* continuously watches *l.from* waiting for the next deliverable signal in *l.from.queue*. Whenever *l* is applicable to a waiting signal *s* (as identified by the head of *l.from.queue*), it attempts to remove *s* from *l.from.queue* in order to insert *s* into *l.to.schedule*. Note that this attempt need not necessarily be successful as, in general, there may be several link agents competing for the same signal *s*.
To avoid that link agents interfere with each other, i.e. two or more link agents attempt to access the same signal *s* simultaneously, an auxiliary dynamic function *access* is introduced (as a kind of control flag), effectively enforcing a synchronisation of link agents as expressed in the rule below.[11]

controlled *access*: *GATE* → *AGENT*

A link agent *l* is a legal choice for the transportation of *s* only, if the following two conditions hold: (1) *s.signalType* ∈ *l.with* and (2) there exists an applicable path connecting *l.to* to some final destination matching with the address information of *s*. Abstractly, this second condition can be expressed using a predicate *Reachable*,

Reachable: *GATE* × *TOARG* → *BOOLEAN*

where *TOARG* refers to the address information as specified by *s.toArg*. The function *Reachable* is specified in full within Section 4.5.1.4 below.

LINK-PROGRAM:

```
if Self.from.queue ≠ empty then
    let s = Self.from.queue.head in
        if Applicable(s) then
            DELETE(s,Self.from)
            INSERT(s, Self.delayedTime, Self.to)
            Self.from.access:= Self
where
    Applicable(s: SIGNALINST): BOOLEAN =def
        s.signalType ∈ Self.with ∧ Reachable(Self.to, s.toArg)
endwhere
```

[11] Without an explicit synchronisation mechanism, multiple access to the same signal *s* would in fact result in illegal behaviour causing a duplication of the original signal *s*. (Recall that such behaviour is not excluded by the semantics of partially ordered runs because of the fact that there is actually no conflict.)

4.5.1.4 Reachability

For the definition of the reachability it is necessary that every gate knows which agent it belongs to. This is achieved using a controlled function as defined below.

controlled *myAgent*: *GATE* → *AGENT*

In order to define the reachability it suffices to state how the recipients of a signal can be accessed. This is done with the following derived function that effectively enumerates all paths that go to an agent.

reachableAgents(*g*: *GATE*): *AGENT*-**set** =$_{\text{def}}$
if *g.myAgent.inport* = *g* **then** { *g.myAgent* }
else ⋃ { *a.to.reachableAgents* | *a* ∈ *AGENT*: *a.from* = *g* }

The definition above states that from the input port of an agent only this agent itself is reachable, and for every other gate the reachable agents are those that are reachable from any outgoing channel. Now it is easy to state when a specific agent is reachable via a specific gate.

Reachable(*gate*: *GATE*, *ag*: *AGENT*): *BOOLEAN* =$_{\text{def}}$
(*ag* = *undefined* ∧ *reachableAgents*(*gate*) ≠ ∅) ∨ *ag* ∈ *reachableAgents*(*gate*)

4.5.1.5 Timers

A particular concise way of modelling timers is by identifying timer objects with corresponding timer signals. More precisely, each *active* timer is represented by a corresponding timer signal in the schedule associated with the input port of the related block instance.

TIMER =$_{\text{def}}$ *Timer-definition*

TIMERINST =$_{\text{def}}$ *RSDLPID* × *TIMER*

The information associated with timers is accessed using the functions defined on *SIGNAL* with their usual meaning, except for *t.toArg*, which has the value *undefined*.

signalType(*s*: *SIGNALINST*): *SIGNAL* =$_{\text{def}}$
if *s* ∈ *PLAINSIGNALINST* **then** *s.plainSignalType* **else** *s*.**s-***TIMER* **endif**
values(*s*: *SIGNALINST*): *VALUE** =$_{\text{def}}$
if *s* ∈ *PLAINSIGNALINST* **then** *s.plainValues* **else** *empty* **endif**
sigSender(*s*: *SIGNALINST*): *AGENT* =$_{\text{def}}$
if *s* ∈ *PLAINSIGNALINST* **then** *s.plainSigSender* **else** *s*.**s-***RSDLPID* **endif**
toArg(*s*: *SIGNALINST*): *TOARG* =$_{\text{def}}$
if *s* ∈ *PLAINSIGNALINST* **then** *s.plainToArg* **else** *undefined* **endif**

Active Timers
To indicate whether a timer instance *t* is active or not, there is a corresponding derived predicate *Active*. The value of *Active*(t) is defined as follows.

Active(*t*: *TIMER*): *BOOLEAN* =$_{\text{def}}$ *t* ∈ *Self.inport.schedule.toSet*

Timer Operations
The macros below model the operations *set* and *reset* on timers as executed by a corresponding RSDL agent. An RSDL **set** will be transformed into a reset operation immediately followed by a set operation.

SETTIMER(*timer*: *TIMER*, *time*: *RSDLTIME*) ≡
INSERT(**mk-***TIMERINST*(*Self*, *timer*), *time*, *Self.inport*)

RESETTIMER(*timer*: *TIMER*) ≡
let *t* = **mk-***TIMERINST*(*Self*, *timer*) **in**
if *Active*(*t*) **then**
DELETE(*t*, *Self.inport*)

4.5.2 RSDL Agents

In this section, the domain *AGENT* is further decomposed into some parts. One part is the domain *LINK* as already introduced. The other two parts are *RSDLAGENT* and *RSDLAGENTSET* which are the other types of active agents in an RSDL specification.

RSDLAGENT $=_{\text{def}}$ *AGENT*
RSDLAGENTSET $=_{\text{def}}$ *AGENT*

Agent sets are only used for initialisation purposes, they are not active afterwards.
Every agent *a* has a set of input gates and a set of output gates attached.

ingates: *AGENT* → *GATE*-**set**
outgates: *AGENT* → *GATE*-**set**

Agent may represent behaviour in the sense of *RSDLAGENT*s. Additionally, their behaviour may be defined by the behaviour of their internal agents. Internal agents communicate with each other and with the outside via their gates. The enclosing structural agent unifies the gates of its internal agents and its own gates, thus enabling communication.
Hierarchical system structures are modelled by means of a static function *owner* defined on agents,

controlled *owner*: *AGENT* → *AGENT*

expressing *structural relations* between agents and their constituent components. More specifically, an agent is considered as *owner* of all those agents which form direct sub agents of this agent.

4.5.2.1 Behaviour of Agents

Every (RSDL) agent gets a behaviour associated with it. This is achieved defining the behaviour of all agents using the function *TheBehaviour*. The collected behaviour is defined as the result of the compilation of the (behavioural parts of the) specification (see also Section 4.6.1).

TheBehaviour: → *BEHAVIOUR*

TheBehaviour: *BEHAVIOUR* $=_{\text{def}}$ *rootNodeAS1.compile*

A special *labelling* of graph nodes is used to model specific control-flow information. Intuitively, node labels relate individual operations of an RSDL agent to transition rules in the resulting RSDL machine model. The effect of state transitions of RSDL agents is then modelled by firing the related transition rules in an analogous order.
Labels are abstractly represented by a static domain *LABEL*. A unary dynamic function *label*, defined on process agents, is used to model dynamic rule selection during the execution phase.

static domain *LABEL*

controlled *label*: *AGENT* → *LABEL*

In a given RSDL machine state, *label* identifies for each RSDL agent the particular rule to be fired by this agent. Moreover, each behavioural syntax construct has a start label indicating the first primitive to be interpreted.

startLabel: *DefinitionAS1* → *LABEL*

Initially the *label* of an agent is set to the *startLabel* of its state transition graph.
The behaviour consists of label-action pairs. The label is used to uniquely identify the action and to represent the current state of the interpretation. For the concrete representation of the actions, we refer to Section 4.5.4.

BEHAVIOUR $=_{\text{def}}$ *PRIMITIVE*-**set**

PRIMITIVE $=_{\text{def}}$ *PRIMLABEL* × *ACTION*
PRIMLABEL $=_{\text{def}}$ *LABEL*

Now we have a single rule for the dynamic behaviour of an RSDL agent.

EXECUTION-PROGRAM:

```
if Self.label = undefined then Self.program := undefined
else choose p: p ∈ TheBehaviour ∧ p.s-PRIMLABEL = Self.label
    EVAL(p.s-ACTION)
```

For the meaning of the macro EVAL, we again refer to Section 4.5.4.

4.5.2.2 Agent Instances

In addition to *Self*, RSDL defines the following default functions on agent instances.

controlled *sender*: *AGENT* → *AGENT*
controlled *parent*: *AGENT* → *AGENT*
controlled *offspring*: *AGENT* → *AGENT*

Finally, each agent instance has its local *input port* at which arriving signals are stored until these signals either are actively received or are discarded. Input ports are modelled as finite sequences of signals in terms of a gate.

controlled *inport*: *AGENT* → *GATE*

4.5.2.3 Undefined Behaviour

In order to model undefined behaviour, we introduce a rule macro

UNDEFINEDBEHAVIOUR ≡
    Self.program := UNDEFINED-BEHAVIOUR-PROGRAM

UNDEFINED-BEHAVIOUR-PROGRAM:

```
// the contents of this is not defined
```

The contents of the program UNDEFINED-BEHAVIOUR-PROGRAM is not specified. Whenever the further behaviour of the system is undefined, the current agent is switched to this program. This local assignment is in fact global as the program UNDEFINED-BEHAVIOUR-PROGRAM could involve setting *program* for all agents.

4.5.3 Signal Processing Primitives

4.5.3.1 Input Operation

For the definition of input operations we define a domain of input signal elements.

INPUTDESC =$_{def}$ *INPUTSIGNAL* × *INPUTVARIABLE** × *INPUTCONTINUE*
INPUTSIGNAL =$_{def}$ *SIGNAL*
INPUTVARIABLE =$_{def}$ *Variable-identifier*
INPUTCONTINUE =$_{def}$ *LABEL*

Input signals are processed with the macro CHECKINPUTSIGNAL below.

```
CHECKINPUTSIGNAL(signalset: INPUTDESC-set, saveset: SIGNAL-set, next: LABEL) ≡
    let s = extract(Self.inport.schedule, saveset) in
        if s ≠ undefined then
            DELETE(s, Self.inport)
            if s.signalType ∈ { sig.s-INPUTSIGNAL | sig ∈ signalset } then
                choose sig: sig ∈ signalset ∧ sig.s-INPUTSIGNAL = s.signalType
                    ASSIGNVALUES(sig.s-INPUTVARIABLE-seq, s.values)
                    Self.sender:= s.sigSender
                    Self.label:= sig.s-INPUTCONTINUE
            else // discard the signal
            endif
        else Self.label:= next
```

```
ASSIGNVALUES(variables: Variable-definition*, values: VALUE*) ≡
    do forall idx: idx ∈ 1 .. values.length
        if variables[idx] ≠ undefined then
            ASSIGN(variables[idx], values[idx], Self)
```

The macro CHECKINPUTSIGNAL above extracts the first signal from the input port that is not saved as indicated by the save signals. If this signal belongs to the signals the entity can consume, then it is deleted from the input port. If there is an explicit transition connected to this signal, this is activated by setting the label appropriately. Otherwise, the signal is discarded, i.e. nothing else happens. If there is no signal extracted, the computation proceeds with the next step.
An auxiliary function *extract* (defined below), seeks the input port *Self.inport.queue* for the next signal *s* that can be processed, returning *undefined*, if no such signal exists.

```
extract: SIGNALINST* × SIGNAL-set → SIGNALINST

extract(seq: SIGNALINST*, save: SIGNAL-set): SIGNALINST =def
        if seq = empty then
            undefined
        else
            if seq.head.signalType ∈ save then
                extract(seq.tail, save)
            else
                seq.head
```

4.5.3.2 Continuous Signal

For the handling of continuous signals, a new domain *CONTINUOUSSIGNAL* is introduced.

```
CONTINUOUSSIGNAL =def CONTINUOUSVALUE × NEXTLABEL
CONTINUOUSVALUE =def LABEL
NEXTLABEL =def LABEL
```

Please find below the semantics for handling continuous signals.

```
DOCONTINUOUS(contset: CONTINUOUSSIGNAL-set, next: LABEL) ≡
    let EnabledSignals = { c ∈ contset: value2bool(currentValue(c.s-CONTINUOUSVALUE, Self)) } in
        if EnabledSignals ≠ ∅ then
            choose c: c ∈ EnabledSignals
                Self.sender:= Self
                Self.label:= c.s-NEXTLABEL
        else
            Self.label:= next
        endif
```

If there is any continuous signal that is enabled, one is selected to continue.

4.5.3.3 Signal Output

A signal output operation causes the creation of a new signal instance. The agent instance initiating the output operation identifies itself as sender of the signal instance by setting a corresponding function *sigSender* defined on signals. In general, there may be one or more output gates of a process to which a signal can be delivered depending on the specified constraint on possible destinations as stated by the value of *TOARG* that is obtained as parameter of an output operation and is assigned to a signal by setting the corresponding function defined on plain signals.
Possible ambiguities are resolved by a non-deterministic choice for a gate which is connected to a path being *compatible* with *TOARG*. In the rule below, this choice is stated in abstract terms using the predicate *Reachable* (cf. Section 4.5.1.4).

SIGNALOUTPUT(*signalName*: *SIGNAL*, *signalValues*: *VALUE**, *signalToArg*: *TOARG*) ≡
choose *g*: *g* ∈ *Self.outgates* ∧ *Reachable*(*g*, *signalToArg*)
extend *PLAINSIGNALINST* **with** *s*
s.plainSignalType:= *signalName*
s.plainValues:= *signalValues*
s.plainToArg:= *signalToArg*
s.plainSigSender:= *Self*
INSERT(*s*, *now*, *g*)

4.5.4 Behaviour Primitives

In this section we present the behaviour primitives of the SAM using the macros defined in the previous section as a basis.
We define the domain *ACTION* for single steps as follows.

ACTION =$_{\text{def}}$ *ANYORDER* ∪ *TASK* ∪ *OUTPUT* ∪ *CREATE* ∪ *SET* ∪ *RESET*
∪ *SKIP* ∪ *STOP* ∪ *DECISION* ∪ *VAR* ∪ *FUNCALL* ∪ *SYSTEMVALUE* ∪ *TIMERACTIVE*
∪ *CHECKINPUT* ∪ *CHECKCONTINUOUS*

A static domain *VALUELABEL* represents labels in *LABEL* at which an RSDL agent can write or read a value. These values can be accessed by a dynamic, controlled function *currentValue*.

VALUELABEL =$_{\text{def}}$ *LABEL*
CONTINUELABEL =$_{\text{def}}$ *LABEL*

controlled *currentValue*: *VALUELABEL* × *AGENT* → *VALUE*

The meaning of the evaluation of an action is defined with the macro EVAL. Note that the subdomains of *ACTION* are pairwise disjoint.

EVAL(*a*: *ACTION*) ≡
if *a* ∈ *ANYORDER* **then** EVALANYORDER(*a*) **endif**
if *a* ∈ *TASK* **then** EVALTASK(*a*) **endif**
if *a* ∈ *OUTPUT* **then** EVALOUTPUT(*a*) **endif**
if *a* ∈ *CREATE* **then** EVALCREATE(*a*) **endif**
if *a* ∈ *SET* **then** EVALSET(*a*) **endif**
if *a* ∈ *RESET* **then** EVALRESET(*a*) **endif**
if *a* ∈ *SKIP* **then** EVALSKIP(*a*) **endif**
if *a* ∈ *STOP* **then** EVALSTOP(*a*) **endif**
if *a* ∈ *DECISION* **then** EVALDECISION(*a*) **endif**
if *a* ∈ *VAR* **then** EVALVAR(*a*) **endif**
if *a* ∈ *FUNCALL* **then** EVALFUNCALL(*a*) **endif**
if *a* ∈ *SYSTEMVALUE* **then** EVALSYSTEMVALUE(*a*) **endif**
if *a* ∈ *TIMERACTIVE* **then** EVALTIMERACTIVE(*a*) **endif**
if *a* ∈ *CHECKINPUT* **then** EVALINPUT(*a*) **endif**
if *a* ∈ *CHECKCONTINUOUS* **then** EVALCONTINUOUS(*a*) **endif**

4.5.4.1 Evaluation in Any Order

Explanation
The any order primitive is used for expressing that some steps can be performed in any order.

Representation

ANYORDER =$_{\text{def}}$ *LABEL*-**set** × *CONTINUELABEL*

controlled *stillToVisit*: *AGENT* → *LABEL*-**set**

Behaviour
The function *stillToVisit* keeps track of all labels that have not been visited. After having visited all labels, this function is reset to *undefined* for a next call of *ANYORDER*. If there are still labels to be visited, one of those is chosen and becomes the next label.

EVALANYORDER(*a*: *ANYORDER*) ≡
  if *Self.stillToVisit* = *undefined* **then** *Self.stillToVisit*:= *a*.**s-***LABEL***-set**
  elseif *Self.stillToVisit* = ∅ **then**
    Self.label:= *a*.**s-***CONTINUELABEL*
    Self.stillToVisit:= *undefined*
  else
    choose *l*: *l* ∈ *Self.stillToVisit*
      Self.label:= *l*
      Self.stillToVisit:= *Self.stillToVisit* \ { *l* }

4.5.4.2 Task

Explanation
The task primitive is used for expressing an assignment.

Representation

TASK =$_{\text{def}}$ *VARIABLENAME* × *VALUELABEL* × *CONTINUELABEL*

Behaviour

EVALTASK(*task*: *TASK*) ≡
    ASSIGN(*task*.s-*VARIABLENAME*, *currentValue*(*task*.s-*VALUELABEL*, *Self*), *Self*)
    Self.label:= *task*.**s-***CONTINUELABEL*

4.5.4.3 Output

Explanation
The output primitive is used for expressing a signal output.

Representation

OUTPUT =$_{\text{def}}$ *SIGNAL* × *VALUELABEL** × *TOARGLABEL* × *CONTINUELABEL*
TOARGLABEL =$_{\text{def}}$ *LABEL*

Behaviour

EVALOUTPUT(*o*: *OUTPUT*) ≡
    SIGNALOUTPUT(*o*.**s-***SIGNAL*, < *currentValue*(*v*, *Self*) | *v* **in** *o*.**s-***VALUELABEL***-seq** >,
      value2agent(*currentValue*(*o*.**s-***TOARGLABEL*, *Self*)))
    Self.label:= *o*.**s-***CONTINUELABEL*

Reference Sections
See also Section 4.5.3.3.

4.5.4.4 Create

Explanation
The create primitive is used for expressing dynamic creation of agent instances at system run time.
Note: There must be a lock that no two instances can be created at the same time (not shown here).

Representation

CREATE $=_{\text{def}}$ *CREATEAGENTDEF* × *CONTINUELABEL*
CREATEAGENTDEF $=_{\text{def}}$ *Agent-definition*

Behaviour

EVALCREATE(*c*: *CREATE*) ≡
    NEWAGENTINSTANCE(*c*.**s-***CREATEAGENTDEF*)
    Self.label:= *c*.**s-***CONTINUELABEL*

NEWAGENTINSTANCE(*agentDesc*: *Agent-definition*) ≡
  choose *a*: *a* ∈ *AGENT* ∧ *a.ref* = *agentDesc*
    let *n* = |{ *inst* ∈ *AGENT*: *inst.owner* = *a* }| **in**
      if *n* < *agentDesc*.**s-***Number-of-instances*.**s-***Maximum-number* **then**
        a.offspring := *Self* // to avoid multiple creation in the same set
        CREATEAGENT(*a*, *Self*)
      else *Self.offspring*:= *nullAgent*

Reference Sections
See also Section 4.6.3.2.

4.5.4.5 Set

Explanation
The set primitive is used for expressing a timer set.

Representation

SET $=_{\text{def}}$ *TIMELABEL* × *TIMERNAME* × *CONTINUELABEL*
TIMELABEL $=_{\text{def}}$ *VALUELABEL*
TIMERNAME $=_{\text{def}}$ *TIMER*

Behaviour

EVALSET(*s*: *SET*) ≡
    SETTIMER(*s*.**s-***TIMERNAME*, *value2time*(*currentValue*(*s*.**s-***TIMELABEL*, *Self*)))
    Self.label:= *s*.**s-***CONTINUELABEL*

Reference Sections
See also Section 4.5.1.5.

4.5.4.6 Reset

Explanation
The reset primitive is used for expressing a timer reset.

Representation

RESET $=_{\text{def}}$ *TIMERNAME* × *CONTINUELABEL*

Behaviour

EVALRESET(*r*: *RESET*) ≡
    RESETTIMER(*r*.**s-***TIMERNAME*)
    Self.label:= *r*.**s-***CONTINUELABEL*

Reference Sections
See also Section 4.5.1.5.

4.5.4.7 Skip

Explanation
The skip primitive is basically a no-op. It is used for instance to model joins.

Representation

SKIP =$_{\text{def}}$ *LABEL*

Behaviour

EVALSKIP(*s*: *SKIP*) ≡
    Self.label:= *s*.**s-***LABEL*

4.5.4.8 Stop

Explanation
The stop primitive is used for expressing the stop action.

Representation

STOP =$_{\text{def}}$ **stop**

Behaviour

EVALSTOP(*s*: *STOP*) ≡
    Self.program:= *undefined*

4.5.4.9 Decision

Explanation
The decision primitive is used for expressing a control flow branching. If none of the alternatives applies, then the further behaviour is undefined.

Representation

DECISION =$_{\text{def}}$ *VALUELABEL* × *ANSWER***-set**

ANSWER =$_{\text{def}}$ *ANSWERVALUE* × *ANSWERCONTINUE*
ANSWERVALUE =$_{\text{def}}$ *LABEL*
ANSWERCONTINUE =$_{\text{def}}$ *LABEL*

Behaviour

EVALDECISION(*d*: *DECISION*) ≡
    let *dval* = *currentValue*(*d*.**s-***VALUELABEL*, *Self*) **in**
    let *avalues* = { *currentValue*(*a*.**s-***ANSWERVALUE*, *Self*) | *a* ∈ *d*.**s-***ANSWER***-set** } **in**
        if *dval* ∈ *avalues* **then**
            choose *a*: *a* ∈ *d*.**s-***ANSWER***-set** ∧ *currentValue*(*a*.**s-***ANSWERVALUE*, *Self*) = *dval*
                Self.label:= *a*.**s-***ANSWERCONTINUE*
        elseif *undefined* ∈ { *a*.**s-***ANSWERVALUE* | *a* ∈ *d*.**s-***ANSWER***-set** } **then**
            choose *a*: *a* ∈ *d*.**s-***ANSWER***-set** ∧ *a*.**s-***ANSWERVALUE* = *undefined*
                Self.label:= *a*.**s-***ANSWERCONTINUE*
        else UNDEFINEDBEHAVIOUR

4.5.4.10 Evaluation of Variables

Explanation
The *VAR* primitive is used for expressing the evaluation of a variable.

Representation

VAR =$_{\text{def}}$ *VARIABLENAME* × *CONTINUELABEL*

Behaviour
The value of *variableName* in the state of the executing agent is determined by the function *eval* and stored at the current label for *Self*.

EVALVAR(*var*: *VAR*) ≡
currentValue(*Self.label*, *Self*) := *eval*(*var*.**s-***VARIABLENAME*, *Self.state*)
Self.label := *var*.**s-***CONTINUELABEL*

4.5.4.11 Evaluation of System Values

Explanation
The system value primitive is used to compute the predefined values of RSDL agents.

Representation

SYSTEMVALUE =$_{\text{def}}$ *VALUEKIND* × *CONTINUELABEL*
VALUEKIND =$_{\text{def}}$ { *nowKind*, *selfKind*, *parentKind*, *offspringKind*, *senderKind* }

Behaviour

EVALSYSTEMVALUE(*sv*: *SYSTEMVALUE*) ≡
let *k* = *sv*.**s-***VALUEKIND* **in**
if *k* = *nowKind* **then**
currentValue(*Self.label*, *Self*):= **selector-***RSDLTIME*(*now*)
elseif *k* = *selfKind* **then**
currentValue(*Self.label*, *Self*):= *agent2value*(*Self*)
elseif *k* = *parentKind* **then**
currentValue(*Self.label*, *Self*):= *agent2value*(*Self.parent*)
elseif *k* = *offspringKind* **then**
currentValue(*Self.label*, *Self*):= *agent2value*(*Self.offspring*)
elseif *k* = *senderKind* **then**
currentValue(*Self.label*, *Self*):= *agent2value*(*Self.sender*)
endif
endlet
Self.label:= *sv*.**s-***CONTINUELABEL*

4.5.4.12 Evaluation of Predefined Functions and Literals

Explanation
The function call primitive is used for evaluating predefined functions and literals.

Representation

FUNCALL =$_{\text{def}}$ *FUNCTIONNAME* × *VALUELABEL** × *CONTINUELABEL*
FUNCTIONNAME =$_{\text{def}}$ *Name*

Behaviour

EVALFUNCALL(*f*: *FUNCALL*) ≡
currentValue(*Self.label*, *Self*) :=
compute(*f*.**s-***FUNCTIONNAME*, < *currentValue*(*l*, *Self*) | *l* **in** *f*.**s-***VALUELABEL*-**seq** >)
Self.label := *f*.**s-***CONTINUELABEL*

Reference Sections
See also Section 4.4.4.

4.5.4.13 Evaluation of Timer Active Expressions

Explanation
The timer active primitive is used for expressing the evaluation of timer active expressions.

Representation

TIMERACTIVE =$_{def}$ *TIMER* × *CONTINUELABEL*

Behaviour

EVALTIMERACTIVE(*t*: *TIMERACTIVE*) ≡
currentValue(*Self.label*, *Self*) := *bool2value*(*Active*(**mk-***TIMERINST*(*Self*, *t*.**s-***TIMERNAME*)))
Self.label := *t*.**s-***CONTINUELABEL*

Reference Sections
See also Section 4.5.1.5.

4.5.4.14 Input Primitives

Explanation
The input primitives are used to model the behaviour of RSDL agents being in a state, i.e. waiting for signals to arrive or for conditions to become true.

Representation

CHECKINPUT =$_{def}$ *INPUTDESC*-**set** × *SAVESIGNAL*-**set** × *CONTINUELABEL*
SAVESIGNAL =$_{def}$ *SIGNAL*

CHECKCONTINUOUS =$_{def}$ *CONTINUOUSSIGNAL*-**set** × *CONTINUELABEL*

Behaviour

EVALINPUT(*i*: *CHECKINPUT*) ≡
CHECKINPUTSIGNAL(*i*.**s-***INPUTDESC*-**set**, *i*.**s-***SAVESIGNAL*-**set**, *i*.**s-***CONTINUELABEL*)

EVALCONTINUOUS(*c*: *CHECKCONTINUOUS*) ≡
DOCONTINUOUS(*c*.**s-***CONTINUOUSSIGNAL*-**set**, *c*.**s-***CONTINUELABEL*)

Reference Sections
See also Section 4.5.3.1 and Section 4.5.3.2.

4.6 RSDL Abstract Machine Programs

Building on the signal flow concepts of the SAM, defined in Section 4.5, the construction of machine models as described below complements the semantic definition by fixing the missing aspects.

4.6.1 Compilation Function

Here we present the function that compiles an RSDL state machine description into an ASM representation. To start with the compilation, we first need a function to find unique labels for a syntactic entity.

monitored *uniqueLabel*: *DefinitionAS1* × *INT* → *LABEL*

For this function, it holds that

constraint ∀ $d_1, d_2 \in$ *DefinitionAS1*: ∀ $i_1, i_2 \in$ *INT*:
uniqueLabel(d_1, i_1) = *uniqueLabel*(d_2, i_2) ⇔ (d_1=d_2 ∧ i_1=i_2)

Now we define the compilation function. The rules below describe two functions for the compilation.

compile: *DefinitionAS1* → *BEHAVIOUR*

compileExpr: *DefinitionAS1* × *LABEL* → *BEHAVIOUR*

The computed value of an expression *e* is always stored at *currentValue*(*uniqueLabel*(*e*,1), *Self*).
The definition of the compilation function is done using a series of patterns and the corresponding results. Afterwards, the function *startLabel* is defined also with a series of patterns in Section 4.6.1.6.

4.6.1.1 States and Triggers

The following parts are considered to form the definition of the function *compile* if put together with the following header. The contents of the case expression are all the compilation cases as given below.

compile(*a*: *DefinitionAS1*): *BEHAVIOUR* =$_{\text{def}}$
case *a* **of**

All the contents of this function is given as patterns and what the result of the function is for these patterns. The default case when no pattern is matching is the collected set of all the results of all children nodes.

| *g*=*State-transition-graph*(*start*, *states*, *freeActions*) =>
compile(*start*) ∪
⋃{ *compile*(*s*) | *s* ∈ *states* } ∪
⋃{ *compile*(*f*) | *f* ∈ *freeActions* }

The compilation of a graph is defined by the compilation of its parts.

| *Start-node*(*trans*) => *compile*(*trans*)

The compilation of a start node is the compilation of its transition.

| *s*=*State-node*(*, *save*, *inputs*, *cont*) =>
{**mk-***PRIMITIVE*(*uniqueLabel*(*s*,1), **mk-***CHECKINPUT*(*insignals*, *save*, *uniqueLabel*(*s*,2))) } ∪
⋃{ *compile*(*i*.**s-***Transition*) | *i* ∈ *inputs* } ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*s*,2), **mk-***ANYORDER*(*expressions*, *uniqueLabel*(*s*,3))) } ∪
⋃{ *compileExpr*(*c*.**s-***Continuous-expression*, *uniqueLabel*(*s*,2)) | *c* ∈ *cont* } ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*s*,3), **mk-***CHECKCONTINUOUS*(*contsignals*, *uniqueLabel*(*s*,1))) } ∪
⋃{ *compile*(*c*.**s-***Transition*) | *c* ∈ *cont* }
where
insignals:*INPUTDESC*-**set** =$_{\text{def}}$ { **mk-***INPUTDESC*(*i*.**s-***Signal-identifier*, *i*.**s-***Variable-identifier*-**seq**,
startLabel(*i*.**s-***Transition*)) | *i* ∈ *inputs* }
contsignals: *CONTINUOUSSIGNAL*-**set** =$_{\text{def}}$
{ **mk-***CONTINUOUSSIGNAL*(*uniqueLabel*(*c*.**s-***Continuous-expression*,1),
startLabel(*c*.**s-***Transition*)) | *c* ∈ *cont* }
expressions: *LABEL*-**set** =$_{\text{def}}$ { *startLabel*(*c*.**s-***Continuous-expression*) | *c* ∈ *cont* }
endwhere

The following order of computation is defined with the above:
1. Check ordinary input signals (*uniqueLabel*(*s*,1)).
2. Compute all continuous signal expressions in arbitrary order (*uniqueLabel*(*s*,2)).
3. Check continuous signals (*uniqueLabel*(*s*,3)).
4. Go back to the beginning.

This is expressed using several calls of the compile function for the several parts. The *insignals* are a set of *INPUTDESC* and the *contsignals* are a set of *CONTINUOUSSIGNAL* as defined in Section 4.5.2.3.

| *Free-action*(*, *trans*) => *compile*(*trans*)

The transformation of a free action is given by the transformation of its transition. The start label of the transition is found within the transformation of the join.

4.6.1.2 Transitions

| *t*=*Transition*(*nodes*, *endnode*) =>
    compileNodes ∪
    compile(*endnode*)
where
    compileNodes: *BEHAVIOUR* =$_{\text{def}}$
        if *nodes* = *empty* **then** ∅
        else *compileExpr*(*nodes.last*, *startLabel*(*endnode*)) ∪
            U{ *compileExpr*(*nodes*[*i*], *startLabel*(*nodes*[*i*+1])) | *i* ∈ 1..*nodes.length* }
endwhere

The compilation of a transition is the compilation of its nodes and the compilation of the endnode. The nodes are linked together using the function *compileExpr* which has a second parameter: the continue label.

4.6.1.3 Terminators

| *n*=*Nextstate-node*(*name*) =>
    { **mk-***PRIMITIVE*(*uniqueLabel*(*n*,1), **mk-***SKIP*(*startLabel*(*name.referstoName1*))) }

The compilation of a nextstate is given by a skip to the corresponding state label.

| *s*= *Stop-node*() => {**mk-***PRIMITIVE*(*uniqueLabel*(*s*,1), **mk-***STOP*()) }

The compilation of a stop node is given by a stop instruction.

| *j*= *Join-node*(*name*) =>
    { **mk-***PRIMITIVE*(*uniqueLabel*(*j*,1), **mk-***SKIP*(*startLabel*(*name.referstoName1*))) }

The compilation of a join node is given by a skip to the corresponding free action.

| *d*=*Decision-node*(*q*, *ans*, *elsepart*) =>
    compileExpr(*q*, *uniqueLabel*(*d*,1)) ∪
    {**mk-***PRIMITIVE*(*uniqueLabel*(*d*,1), **mk-***ANYORDER*(*answerExpr*, *uniqueLabel*(*d*,2))) } ∪
    U{ **U**{ *compileExpr*(*e*, *uniqueLabel*(*d*,1)) | *e* ∈ *a*.**s-***Constant-expression*-**set** } | *a* ∈ *ans* } ∪
    {**mk-***PRIMITIVE*(*uniqueLabel*(*d*,2), **mk-***DECISION*(*uniqueLabel*(*q*,1), *answers*)) } ∪
    if *elsepart* ≠ *undefined* **then** *compile*(*elsepart*.**s-***Transition*) **else** ∅ **endif** ∪
    U{ *compile*(*a*.**s-***Transition*) | *a* ∈ *ans* }
where
    answers: *ANSWER*-**set** =$_{\text{def}}$
        U{ { **mk-***ANSWER*(*uniqueLabel*(*e*,1), *startLabel*(*a*.**s-***Transition*)) |
                e ∈ *a*.**s-***Constant-expression*-**set** } | *a* ∈ *ans* } ∪
        if *elsepart* ≠ *undefined* **then** { **mk-***ANSWER*(*undefined*, *startLabel*(*elsepart*.**s-***Transition*)) }
        else ∅ **endif**
    answerExpr: *LABEL*-**set** =$_{\text{def}}$
        U{ { *startLabel*(*e*) | *e* ∈ *a*.**s-***Constant-expression*-**set** } | *a* ∈ *ans* }
endwhere

The compilation of a decision generates the following parts:

1. evaluate the question.
2. evaluate the answers in any order.
3. branch to a matching answer using the primitive *DECISION*.

This concludes the definition of the *compile* function.

endcase // end of the compile function definition

4.6.1.4 Actions

The following compilation parts define the function *compileExpr* with the following header.

compileExpr(*a*: *DefinitionAS1*, *next*: *LABEL*): *BEHAVIOUR* =$_{\text{def}}$
    case *a* **of**

All the contents of this function is given as patterns and what the result of the function for these patterns is. The default result when no pattern is matching is the empty set. All the patterns given below may use the variable *next* referring to the next label to process.

| *a=Assignment*(*id*, *expr*) =>
compileExpr(*expr*, *uniqueLabel*(*a*,1)) ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*a*,1), **mk-***TASK*(*id*, *uniqueLabel*(*expr*,1), *next*)) }

The compilation of an assignment is given by the task primitive.

| *o=Output-node*(*sig*, *expr*, *dest*) =>
if *dest* = *undefined* **then** ∅ **else** *compileExpr*(*dest*, *firstlabel*) **endif** ∪
if *expr* = *empty* **then** ∅
else *compileExpr*(*expr.last*, *uniqueLabel*(*o*,1)) ∪
U{ *compileExpr*(*expr*[*i*], *startLabel*(*expr*[*i*+1])) | *i* ∈ 1..*length*(*expr*)-1 }
endif ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*o*,1),
mk-*OUTPUT*(*sig*, *values*, *uniqueLabel*(*dest*, 1), *next*)) }
where
values: *LABEL** =$_{\text{def}}$ < *uniqueLabel*(*e*,1) | *e* **in** *expr* >
firstlabel: *LABEL* =$_{\text{def}}$
if *expr* = *empty* **then** *uniqueLabel*(*o*,1) **else** *startLabel*(*expr.head*)
endwhere

The compilation of an output proceeds with the following steps:

1. First the destination expression is evaluated.
2. The parameter expressions are evaluated in their order.
3. The output primitive is used to generate a new output signal.

| *c=Create-request-node*(*agent*) =>
{**mk-***PRIMITIVE*(*uniqueLabel*(*c*,1), **mk-***CREATE*(*agent.refersto1*, *next*)) }

The create is compiled into the create primitive.

| *s=Set-node*(*when*, *what*) =>
compileExpr(*when*, *uniqueLabel*(*s*,1)) ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*s*,1), **mk-***RESET*(*what*, *uniqueLabel*(*s*,2))) } ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*s*,2), **mk-***SET*(*uniqueLabel*(*when*,1), *what*, *next*)) }

The compilation of a set proceed with first the evaluation of the time expression followed by a reset primitive and then the set primitive.

| *r=Reset-node*(*id*) => {**mk-***PRIMITIVE*(*uniqueLabel*(*r*,1), **mk-***RESET*(*id*, *next*)) }

A reset is compiled to the reset primitive.

4.6.1.5 Expressions

| *l=Literal*(*name*) =>
{**mk-***PRIMITIVE*(*uniqueLabel*(*l*,1), **mk-***FUNCALL*(*name*, *empty*, *next*)) }

A literal is evaluated using the function call primitive.

| *e=Operation-application*(*name*, *args*) =>
compileExpr(*args.last*, *uniqueLabel*(*e*,1)) ∪
if *args.last* = *args.head* **then** ∅ **else** *compileExpr*(*args.head*, *startLabel*(*args.last*)) **endif** ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*e*,1), **mk-***FUNCALL*(*name*, < *uniqueLabel*(*a*,1) | *a* **in** *args* >, *next*)) }

An operation application is evaluated using the function call primitive. The parameters are evaluated first in their order. Please note that we only have operations with one or with two arguments.

| *id=Identifier*(*,*) => {**mk-***PRIMITIVE*(*uniqueLabel*(*id*,1), **mk-***VAR*(*id*, *next*)) }

A variable access is compiled to the variable primitive.

| *n=Now-expression*() => {**mk-***PRIMITIVE*(*uniqueLabel*(*n*,1), **mk-***SYSTEMVALUE*(*nowKind*, *next*)) }

| *s=Self-expression*() => {**mk-***PRIMITIVE*(*uniqueLabel*(*s*,1), **mk-***SYSTEMVALUE*(*selfKind*, *next*)) }

| *p=Parent-expression*() => {**mk-***PRIMITIVE*(*uniqueLabel*(*p*,1), **mk-***SYSTEMVALUE*(*parentKind*, *next*)) }

| *o=Offspring-expression*() =>
{**mk-***PRIMITIVE*(*uniqueLabel*(*o*,1), **mk-***SYSTEMVALUE*(*offspringKind*, *next*)) }

| *s=Sender-expression*() => {**mk-***PRIMITIVE*(*uniqueLabel*(*s*,1), **mk-***SYSTEMVALUE*(*senderKind*, *next*)) }

The system value enquiries are represented by the system value primitive.

| *t*=*Timer-active-expression*(*id*) => {**mk-***PRIMITIVE*(*uniqueLabel*(*t*,1), **mk-***TIMERACTIVE*(*id*, *next*)) }

A timer active expression is represented by a timer active primitive.

This concludes the definition of the expression compilation function.

endcase // end of the compileExpr function definition

4.6.1.6 Start Labels

This section introduces the function *startLabel* which is responsible to define the start labels of all behavioural syntax constructs.

startLabel(*x*: *DefinitionAS1*): *LABEL* =$_{\text{def}}$
case *x* **of**
| *g*=*Agent-type-definition*(*, *, *, *, *, *, *, *, *graph*) =>
if *graph* ≠ *undefined* **then** *startLabel*(*graph*) **else** *undefined* **endif**
| *g*=*State-transition-graph*(*start*, *, *) => *startLabel*(*start*)
| *Start-node*(*trans*) => *startLabel*(*trans*)
| *s*=*State-node*(*, *, *, *) => *uniqueLabel*(*s*,1)
| *Free-action*(*, *trans*) => *startLabel*(*trans*)
| *Transition*(*nodes*, *endnode*) =>
if *nodes* = *empty* **then** *startLabel*(*endnode*) **else** *startLabel*(*nodes.head*) **endif**
| *n*=*Nextstate-node*(*) => *uniqueLabel*(*n*,1)
| *s*= *Stop-node*() => *uniqueLabel*(*s*,1)
| *j*= *Join-node*(*name*) => *uniqueLabel*(*j*,1)
| *Decision-node*(*q*, *, *) => *startLabel*(*q*)
| *Assignment*(*, *expr*) => *startLabel*(*expr*)
| *o*= *Output-node*(*, *expr*, *dest*) =>
if *dest* ≠ *undefined* **then** *startLabel*(*dest*)
elseif *expr* = *empty* **then** *uniqueLabel*(*o*,1)
else *startLabel*(*expr.head*) **endif**
| *c*=*Create-request-node*(*) => *uniqueLabel*(*c*,1)
| *Set-node*(*when*, *) => *startLabel*(*when*)
| *r*=*Reset-node*(*) => *uniqueLabel*(*r*,1)
| *l*=*Literal*(*) => *uniqueLabel*(*l*,1)
| *Operation-application*(*, *args*) => *startLabel*(*args.head*)
| *v*=*Identifier*(*, *) => *uniqueLabel*(*v*,1)
| *n*=*Now-expression*() => *uniqueLabel*(*n*,1)
| *s*=*Self-expression*() => *uniqueLabel*(*s*,1)
| *p*=*Parent-expression*() => *uniqueLabel*(*p*,1)
| *o*=*Offspring-expression*() => *uniqueLabel*(*o*,1)
| *s*=*Sender-expression*() => *uniqueLabel*(*s*,1)
| *t*=*Timer-active-expression*(*) => *uniqueLabel*(*t*,1)
endcase

4.6.2 Pre-Initial System State

This section states requirements on the initial states S_0 of the abstract RSDL machine model. Initially, there is a single agent *system* denoting a uniquely determined system instance from the domain *RSDLAGENTSET*.

static *system*: → *AGENT*

initially *AGENT* = { *system* }
initially *system.ref* = *rootNodeAS1*.**s-***Agent-definition*
initially *system.mode* = *initial*
initially *system.owner* = *undefined*
initially *system.program* = INIT-AGENT-SET-PROGRAM

The initial system agent has not many functions defined. Most of the other functions are initialised during the initialisation phase or are derived functions. The mode is used to distinguish between two phases of the initialisation as described below.

4.6.3 System Initialisation

Starting from S_0, the initialisation rules describe a recursive *unfolding* of the specified system instance according to its hierarchical structure. Each initialisation step may create several object instances simultaneously.
During the initialisation phase, agents may operate in two different operation modes as stated by a static domain *MODE*,

MODE $=_{\text{def}}$ {*initial*, *starting* }

where a dynamic function *mode* defined on agents indicates the mode of an agent in a given abstract machine state.

controlled *mode*: *AGENT* → *MODE*

Furthermore, two derived function *ingates* and *outgates* are introduced that collect all input gates and all output gates of an agent. This is achieved using the link of the gates to their owning agent. This function is only defined for the RSDL agent sets, the RSDL agents use the gates of their agent sets.

ingates(*a*: *AGENT*): *GATE*-**set** $=_{\text{def}}$
if *a.ref* ≠ *undefined* **then** { *g* ∈ *GATE*: *g.myAgent* = *a* ∧ *g.direction* = *inDir* }
else *Self.owner.ingates* **endif**

outgates(*a*: *AGENT*): *GATE*-**set** $=_{\text{def}}$
if *a.ref* ≠ *undefined* **then** { *g* ∈ *GATE*: *g.myAgent* = *a* ∧ *g.direction* = *outDir* }
else *Self.owner.outgates* **endif**

A unary function *ref* defined on agents identifies for each agent an abstract syntax tree (sub-) definition to be processed during the initialisation phase. A similar reference to then definition is introduced for gates (*gateRef*). A derived function *myType* refers to the type definition of the agent set.

controlled *ref*: *AGENT* → *DefinitionAS1*
controlled *gateRef*: *GATE* → *DefinitionAS1*

myType(*a*: *AGENT*): *DefinitionAS1* $=_{\text{def}}$
if *a.ref* ≠ *undefined* **then** *a.ref*.**s-***Agent-type-identifier.refersto1* **else** *undefined* **endif**

4.6.3.1 Agent Set Initialisation

The initialisation of agent sets (and hence also of systems) is defined by the macro below, where *Self* initially denotes the distinguished system agent *system*.
The rule below is performed once, afterwards the agent is not active any more, because its *program* is *undefined*. However, the mode is set to *starting* because the internal agents have been created.

INIT-AGENT-SET-PROGRAM:

```
CREATEGATES(Self, Self.myType.s-Gate-definition-set)
do forall i: i ∈ 1 .. Self.ref.s-Number-of-instances.s-Initial-number
    CREATEAGENT(Self, nullAgent)
enddo
Self.program := undefined
Self.mode := starting
```

4.6.3.2 Agent Creation

For the creation of an agent, the owner agent set has to be known as well as the creating parent. The predefined RSDL system variables are initialised to be “null” and an input port (a gate) for the agent is created. For agents also the same handling with the mode is used.

```
CREATEAGENT(Owner: AGENT, Parent: AGENT) ≡
    extend AGENT with a
        if Parent ≠ nullAgent then Parent.offspring := a endif
        a.owner := Owner
        a.parent := Parent
        a.sender := nullAgent
        a.offspring := nullAgent
        extend GATE with g
            g.schedule := empty
            a.inport := g
        endextend
        a.mode := initial
        a.program := INIT-AGENT-PROGRAM
```

4.6.3.3 Agent Initialisation

In order to prevent the initialisation rule from being performed more than once, the *mode*-function is used. This function ensures in this case, that the rule below is executed exactly twice, namely once with the value of *Self.mode = initial* and the second time with *Self.mode = starting*. Within the second execution of the rule all sub agent sets are checked to be out of the *initial* mode. This is necessary for all gates to be created when the channels are created.

INIT-AGENT-PROGRAM:

```
if Self.mode = initial then
    // Execute this subrule only once
    Self.mode := starting
    Self.label := Self.owner.myType.startLabel
    Self.state:= initAgentState(Self.owner.myType.s-Variable-definition-set)
    do forall sas: sas ∈ Self.owner.myType.s-Agent-definition-set
        CREATEAGENTSET (Self, sas)
else if initial ∉ { a.mode | a ∈ AGENT: a.owner = Self } then
    CREATECHANNELS(Self.owner.myType.s-Channel-definition-set)
    // switch to execution program
    Self.program := EXECUTION-PROGRAM
    do forall g: g ∈ Self.owner.ingates
        CREATELINK(g, Self.inport, undefined)
    enddo
endif
```

4.6.3.4 Agent Set Creation

The creation of an agent set is done like the creation of the initial system agent set.

```
CREATEAGENTSET(Owner: AGENT, AgentDefinition: DefinitionAS1) ≡
        extend AGENT with sas
            sas.ref := AgentDefinition
            sas.mode := initial
            sas.owner := Owner
            sas.program := INIT-AGENT-SET-PROGRAM
```

4.6.3.5 Channel and Link Creation

Channels are modelled through unidirectional channel paths. Each channel path is represented by an agent of type *LINK*. Creation of channels is given by the creation of their channel paths.

```
CREATECHANNELS(ChannelDefSet: Channel-definition-set) ≡
        do forall item: item ∈ ChannelDefSet
            do forall path: path ∈ item.s-Channel-path-set
                CREATECHANNELPATH(path)
```

The creation of a channel path means to create a new *LINK* agent. the problem is to correctly set the input and the output gates. The detection of the correct gate is done by the properties of the gate. If all runs correctly, the choose runs over a set with exactly one element.

CREATECHANNELPATH(*Path*: *Channel-path*) ≡
  choose *gIn*: *gIn* ∈ *GATE* ∧ *gIn.gateRef* = *Path*.**s-***Originating-gate* ∧
    (*OwnGate*(*gIn*, *inDir*) ∨ *EnclosedGate*(*gIn*, *outDir*))
  choose *gOut*: *gOut* ∈ *GATE* ∧ *gOut.gateRef* = *Path*.**s-***Destination-gate* ∧
    (*OwnGate*(*gOut*, *outDir*) ∨ *EnclosedGate*(*gOut*, *inDir*))
    CREATELINK(*gIn*, *gOut*, *Path*)
where
  OwnGate(*g*: *GATE*, *dir*: *DIRECTION*): *BOOLEAN* =$_{\text{def}}$ *g.myAgent* = *Self.owner* ∧ *g.direction* = *dir*
  EnclosedGate(*g*: *GATE*, *dir*: *DIRECTION*): *BOOLEAN* =$_{\text{def}}$ *g.myAgent.owner* = *Self* ∧ *g.direction* = *dir*
endwhere

The creation of a link is given by creating a new *LINK* agent with the LINK-PROGRAM, the corresponding gates and the signal list for the function *with*. There are two places where links are created: for channel paths and for the connection of an agent to the gate of its parent.

CREATELINK(*fromGate*: *GATE*, *toGate*: *GATE*, *path*: *CHANNEL*) ≡
  extend *AGENT* **with** *l*
    l.program := LINK-PROGRAM
    l.from := *fromGate*
    l.to := *toGate*
    l.channel := *path*

4.6.3.6 Gate Creation

The creation of a gate is split in two parts, namely creation of a gate for each of the two directions, if present.

CREATEGATES(*Owner*: *AGENT*, *GateDefSet*: *Gate-definition*-**set**) ≡
  do forall *item*: *item* ∈ *GateDefSet*
    if *item*.**s-***In-signal-identifier*-**set** ≠ ∅ **then**
      extend *GATE* **with** *g*
        g.myAgent:= *Owner*
        g.gateRef:= *item*
        g.direction:= *inDir*
    endif
    if *item*.**s-***Out-signal-identifier*-**set** ≠ ∅ **then**
      extend *GATE* **with** *g*
        g.myAgent:= *Owner*
        g.gateRef:= *item*
        g.direction:= *outDir*
    endif
  enddo

Part 5: RSDL REFERENCE IMPLEMENTATION

The aim of the reference implementation is to provide an implementation of the formal RSDL semantics that is able to be used for deciding uncertain questions in the scope of the RSDL semantics. It shall be used as a quick reference to the formal model. So anyone can use this reference implementation without knowledge of the formal model and even without knowing RSDL. This reference implementation is not intended to be a particularly effective implementation, instead it is intended to be a particularly precise implementation in that it exactly matches the formal definition. So it will be used for the compilation of small sample specifications and their interpretation thereafter. The handling of large specifications and code generation issues are left to the industrial tools.

Please look again at Figure 4 in Part 1 for an overview of the implementation of the semantics. It has to be noted, that the implementation tooling follows the methodology as introduced in Section 2.2. Moreover, also the result of the implementation follows the methodology. So the methodology is used to generate files that themselves also follow the methodology guidelines. From these generated files the RSDL compiler is produced. The compiler will be built using the following parts in accordance with the language description.

1. A lexical analysis as given by the generated file `rsdl-lexic.l`.
2. A syntax analysis as given by the generated file `rsdl-cs.y`.
3. A concrete syntax tree description (AS0) as given by the generated file `rsdl-as0.k`.
4. An abstract syntax tree description (AS1) as given by the generated file `rsdl-as1.k`.
5. The transformations over the AS0 as given by the generated file `rsdl-trans.k`.
6. Well-formedness rules for the AS0 as given by the generated file `rsdl-cond0.k`.
7. Well-formedness rules for the AS1 as given by the generated file `rsdl-cond1.k`.
8. The representation of the auxiliary functions as given by the generated file `rsdl-fun.k`.
9. The various syntax selection functions are included in the generated file `rsdl-select.k`.
10. The mapping from AS0 to AS1 as given by the generated file `rsdl-map.k`.
11. The compilation of AS1 as given by the generated file `rsdl-compile.k`.
12. The output of the AS1 tree to an ASM representation as given by the generated file `rsdl-toASM.k`.
13. The RSDL runtime system includes a textual version of the ASM parts (SAM, initialisation, data) in the generated file `rsdl-asm.asm`.

Unfortunately, it is impossible to present the complete implementation within this book. Instead, we will concentrate on the major aspects and the explanation of the key concepts in the implementation. In particular the BNF related parts are explained in more depth. Moreover, the full implementation can be found in the internet in [26]. This version will be continuously updated if bugs in the current implementation or in the semantics description are discovered. Moreover, there is also the implementation for the full SDL to be found at the same place in case this is wanted. Please find the file structure of the implementation in the next chapter. Afterwards, the single steps as listed above are described in more detail.

5.1 File Structure of the Implementation

The following table lists the main parts of the implementation that are represented as directories.

Directory Name	Description
RSDLC	The generated RSDL compiler is put together in the directory RSDLC. See also Section 5.6.
RSDLC/Inputs	The implementation is based on extracted files from the textual description of the language and of the semantics. These files are stored here. See also Section 5.2.
RSDLC/Syntax	The Syntax directory contains the handling of the lexis, the concrete syntax, the abstract syntax level 0 (AS0) and the abstract syntax (AS1). See also Section 5.3
RSDLC/ASM	The ASM directory contains the front-end part of all semantics files as well as the handling of the SAM, the initialisation, the data and the compilation. See also Section 5.4.
RSDLC/Satanic	The Satanic directory contains the back-end handling of the transformations, the conditions, the mapping and the auxiliary functions. See also Section 5.5.
RSDLC/Runtime	The Runtime directory includes all the runtime support for RSDL as generated by the RSDLC, by the ASM tooling and by the Syntax tooling. See also Section 5.7.

The files belonging to the directories listed above are listed and explained in the following subsections.

5.2 Extraction of the Files

The whole book is written using Microsoft Word. This had to be taken into account for the extraction of the RSDL language description files. To enable an easy extraction, dedicated styles have been used for setting the different formal language description parts. This makes it possible to use Microsoft Word macros to extract the texts. However, some transformations have to be performed in order to keep all necessary information when saving the extracted text as plain text. The character formats as well as the special symbols have to be replaced by a textual representation that is visible with ASCII characters. For the representation of character formats a prefix is introduced at the place where the corresponding format starts. See the list of available prefixes with their meaning below.

Prefix	Meaning
d-	domain name
f-	function name
a-	ASM parameter or variable name
kw-	Keyword
p-	program name
r-	rule macro name

Please note that no prefixes are used for the concrete syntax symbols, because they can be distinguished by their lexical structure from other names.
For special symbols, a LaTeX-like notation is used as can be seen in the table below.

Special symbol	Representation	Special symbol	Representation
∀	\forall	⊂	\subset
∃	\exists	⊆	\subseteq
≤	\leq	∈	\in
≥	\geq	∉	\notin
→	\rightarrow	¬	\not
×	\times	∧	\land
≠	\neq	∨	\lor
≡	\equiv	⇔	\iff
∅	\emptyset	⇒	\implies
∩	\intersection	⌢	\concat
∪	\union	⋃	\bigunion

The following files in the directory `Input` are extracted using the methods as explained above.

File name	Description
rsdl-lexic.txt	Extracted lexis as defined in the language definition.
rsdl-cs-extr.txt	Extracted concrete syntax as defined in the language definition.
rsdl-as1.txt	Extracted abstract syntax as given in the language definition.
sem-as0.txt	Extracted AS0 syntax as given in the formal semantics.
sem-as1.txt	Extracted abstract syntax as given in the formal semantics.
sem-trans.txt	Extracted transformations as given in the formal semantics.
sem-cond0.txt	Extracted conditions on AS0 as given in the formal semantics.
sem-cond1.txt	Extracted conditions on AS1 as given in the formal semantics.
sem-map.txt	Extracted mapping from AS0 to AS1 as given in the formal semantics.
sem-compile.txt	Extracted compilation function as given in the formal semantics.
sem-fun.txt	Extracted auxiliary functions as given in the formal semantics.
sem-asm.txt	Extracted ASM parts (SAM, data and initialisation) as given in the formal semantics.

Moreover, there are some predefined files in the directory `Input`.

File name	Description
rsdl-cs.txt	This file is manually derived from the file `rsdl-cs-extr.txt`. It includes the unambiguous RSDL grammar (see 4.2).
as0.tok	This file includes the tokens that are used by the AS0 grammar.
as1.tok	This file includes the tokens that are used by the AS1 grammar.

The intention in the extraction process is to make it as unintelligent as possible, i.e. to only extract information that was already in the original text and not to start already with transforming the text. If the text would have been written with TeX or LaTeX, the input could have been used as is. The intelligence in handling the text is embedded into the post-processing tools as described in the remainder of this part of the book.

5.3 Implementation of the Syntax Representations

There are several formal parts that in fact represent syntax descriptions. These are the following extracted files: `rsdl-lexic.txt`, `rsdl.cs-extr.txt`, `rsdl-cs.txt`, `rsdl-as1.txt`, `sem-as0.txt` and also `sem-as1.txt`. The handling of these files is implemented in the directory `Syntax`.

5.3.1 Overall Overview and makefile

Within the syntax implementation we make heavy use of file extensions. We introduce different extensions for the various kinds of files and formulate the dependencies between the types as dependencies between the extensions.

A warning is appropriate at this place. The syntax implementation talks about tools that process syntax descriptions. However, these tools are also written according to the methodology description in Section 2.2. Therefore they also use an abstract syntax representation (for the representation of the RSDL syntax) and have lexical rules etc. In order to reduce the risk of misunderstanding, the following conventions will be used within Section 5.3. The language for formulating the RSDL syntax is called BNF with the variants lexis BNF, concrete syntax BNF, abstract syntax BNF and AS0 BNF. The representation of this syntax within the syntax tooling is denoted by postfixes to BNF, e.g. BNF abstract syntax is the abstract syntax for BNF.

As there are several formats that are all represented by the same BNF abstract syntax structure, we distinguish between generation of the BNF abstract syntax tree (front-end tools) and working on the BNF abstract syntax tree (back-end tools). The back-end tools have all the same behaviour, which is defined by the file `backend.c`. The back-end handling only means to call the unparse view and the rewrite views as determined from the name of the executable.

We present in the following the general parts of the `makefile` as far as applicable for all parts. Please note that much of this handling was already explained in Section 2.3.4.

The first part of the `makefile` are general flags and variables.

```
WINNT=1
# Tools
ifndef WINNT
SHELL   = /bin/sh
endif
YACC    = bison
CC      = gcc -g
KC      = kc4
LEX     = flex

ifdef WINNT
EXE = .exe#     # postfix for executables
endif

# Flags
YFLAGS  = -d -y
LFLAGS  = -t
CFLAGS  = -Wall -DYYDEBUG -DYYERROR_VERBOSE

# directories
INPUTS      = ../Inputs
OUTPUTS     = ../SDLC
```

The second part of the makefile are file name prefixes for the individual files.

```
# document prefixes
RSDL    = rsdl
SEM     = sem
RSDL_L  = ${RSDL}-lexic   # the lexis description
RSDL_C  = ${RSDL}-cs      # the concrete syntax description
RSDL_0  = ${RSDL}-as0     # the AS0 description
RSDL_1  = ${RSDL}-as1     # the AS1 description
RSDL_O  = ${RSDL_C}-extr  # the extracted concrete syntax
SEM_0 = ${SEM}-as0        # the AS0 from the semantics part
SEM_1 = ${SEM}-as1        # the AS1 from the semantics part
```

The next part defines the structure of the code.

```
# Sources
KFILES  = syn-abstract.k syn-semantics.k syn-pretty.k \
          syn-genlex.k syn-gentoken.k syn-genyacc.k \
          syn-cst2ast.k syn-genk.k syn-gentxt.k syn-genasm.k \
          syn-gensatanictoken.k syn-gensatanicsel.k

YFILE   = syntax.y
COBJ    = functions.o
FRONTO  = frontend.o
EXTRA   = ${FRONTO} ${COBJ}
BACKEND = backend
OUTPUTFILES = ${RSDL_L}.l ${RSDL_C}.y ${RSDL_0}.k ${RSDL_1}.k ${RSDL}-toASM.k
SATANICFILES = ${RSDL_1}.satsel.cc ${RSDL_0}.satsel.cc ${RSDL_L}.sattok.cc
RUNTIMEFILES = ${SEM_1}.asm
```

Now the programs to be generated are defined. They are distinguished between front-end processing and back-end processing.

```
# Programs
FRONTENDS   = lex2ast${EXE} cs2ast${EXE} as12ast${EXE}
BACKENDS    = ast2pretty${EXE} ast2l${EXE} ast2tok${EXE} \
              ast2gst${EXE} gst2kst${EXE} \
              ast2one${EXE} gst2y${EXE} ast2k${EXE} kst2txt${EXE}
```

Now we can define the global targets.

Target	Description
lex	Generate the parts belonging to the lexis BNF.
cs	Generate the parts belonging to the concrete syntax BNF.
as0	Generate the parts belonging to the AS0 BNF.
as1	Generate the parts belonging to the AS1 BNF.
output	Generate the output files for construction of the RSDL compiler.
runtime	Generate the ASM representation of the AS1 structure.
depend	Generate the dependencies between C files and their header files.
clean	Delete all generated files.

The rules for the targets in the list are given below. The rule for cleaning is omitted here and the rules for the dependency handling are explained further down.

```
.PHONY: notknown lex cs as0 as1 depend clean output

# default rule
notknown: ; @echo "try make { lex | cs | as0 | as1 | depend | clean | output }"

lex: ${RSDL_L}.l ${RSDL_L}.tok
cs: ${RSDL_C}.output cs.diff
as0: as0.diff ${RSDL_C}.y ${RSDL_0}.k
as1: as1.diff ${RSDL_1}.k
output: ${OUTPUTFILES:%=${OUTPUTS}/%} ${SATANICFILES:%=${SATANIC}/%}
runtime: ${RUNTIMEFILES} ${RUNTIMEFILES:%=${RUNTIME}/%}
```

The next part defines the handling of `kimwitu`, `lex` and `yacc`. Please note that there are already many predefined rules built-in into `make` that handle `lex` and `yacc` files.

```
# Auxiliary files
KC_TIME = .kc_time_stamp
KC_OGEN = k.o csgiok.o unpk.o rk.o
KC_OSRC = ${KFILES:.k=.o}
KOBJS   = ${KC_OGEN} ${KC_OSRC} ${COBJ}

# Kimwitu compilation (note: kc does not touch unchanged files)
${KC_TIME}: ${KFILES}
     ${KC} ${KFILES}
     date > ${KC_TIME}

# the normal lex/yacc header trick
${YFILE:.y=.h} : y.tab.h; -cmp -s $@ $< || cp $< $@

y.tab.h : ${YFILE:.y=.c}

%.output: %.y; ${YACC} -v $< || (rm $@; exit 1)
```

Now we define how the programs are made.

Front-end programs depend on their corresponding BNF `lex` file. They all use the same BNF abstract tree and the same BNF `yacc` file. Please find the description of the common front-end main program in Section 5.3.1.1. There is only one back-end program called `backend`. It is used by linking it to another file name which effectively identifies an unparse view and/or rewrite views. These views are then applied with the `backend` program. Please find the description of the `backend` main program below.

```
# How to make the programs
${FRONTENDS}: %${EXE}: ${KC_TIME} ${FRONTO} %-lex.o ${YFILE:.y=.o} ${KOBJS}
     ${CC} ${CFLAGS} -o $*${EXE} ${KOBJS} ${YFILE:.y=.o} $*-lex.o ${FRONTO}

${BACKEND}${EXE}: ${KC_TIME} ${BACKEND}.o ${KOBJS}
     ${CC} ${CFLAGS} -o $@ ${KOBJS} ${BACKEND}.o

${BACKENDS}: ${BACKEND}${EXE}; ln -f $< $@
```

The next part describes how to generate output files. Please note, that the output files are only copied to the output directory when they are really different from the files already there.

```
${OUTPUTS}/%: %; -cmp -s $@ $< || cp $< $@
```

The following part handles the automatic detection and inclusion of the C header file dependencies.

```
depend:  ${KC_TIME} ${FRONTENDS:%${EXE}=%-lex.c} ${YFILE:.y=.c} ${YFILE:.y=.h}
     ${CC} -MM *.c > .depend
     @echo .depend is included

DEPEND=${wildcard .depend}

ifneq "${DEPEND}" ""
include .depend
else
Makefile: MM
MM: ; @echo "**************** You must make depend first ******************"
endif
```

The remaining parts of the `make` handling are explained in the overview places of the next sections.

5.3.1.1 Generic Front End Program

Please find below the generic front end program. The different input versions are only distinct with respect to their lexical structure, the other things are equal. In order to have a useful output for the different parts an identifying string `lexkind` is used.

```
1.  #include <stdio.h>
2.  #include "k.h"
3.  #include "rk.h"
4.  #include "unpk.h"
5.  #include "csgiok.h"
6.  #include "syn-semantics.h"
7.  #include "functions.h"
8.  #include "frontend.h"

9.  spec TheSpec; /* The syntax tree root */

10. int main()
11. { fprintf(stderr,"Transforming %s to AST\n", lexkind);
12.   if (!yyparse())
13.   { init_symtab();
14.     unparse_spec( TheSpec, dummy_printer, create_symtab );
15.     TheSpec = rewrite_spec( TheSpec,basic_rewrite );
16.     unparse_spec( TheSpec, dummy_printer, check_symtab );
17.     CSGIOwrite_spec(stdout, TheSpec);
18.     return 0;
19.   } else return 1;
20. } /* main */
```

The following steps are performed by the front-end tooling.

1) Check the lexical and syntax structure of the input (line 12).
2) Initialise the symbol table (line 13).
3) Fill all symbol defining and using occurrences into the symbol table (line 14).
4) Do some basic rewriting of the BNF abstract tree (line 15).
5) Check the symbol table entries for consistency (line 16).
6) Output the BNF abstract tree (line 17).

5.3.1.2 Generic Back-end Handling

Please find below the generic back-end program. This program first reads in the current abstract syntax tree. Afterwards it looks for rewrite views matching its own executable name. All of these views are used for rewriting one by one. Afterwards an unparse view as the name of the executable is searched. If there is one, it is used for generating the output. Otherwise, the resulting tree is output in the kimwitu internal `CSGIO` format.

```
1.  #include <stdio.h>
2.  #include <string.h>

3.  #include "k.h"
4.  #include "rk.h"
5.  #include "unpk.h"
6.  #include "csgiok.h"
7.  #include "functions.h"
8.  #include "syn-semantics.h"

9.  /* The spec tree root */
10. spec TheSpec;

11. char *errors;

12. int main(int argc, char *argv[])
13. { KC_Printer printer(printer_f);
14.   int i, cnt=0, uv=0; char myname[100], *hlp, myrname[100]="r_";
15.   rview rv=base_rview;

16.   hlp=strrchr(argv[0],'/');
17.   strcpy(myname,hlp?hlp+1:argv[0]);
18.   hlp=strrchr(myname,'.');
19.   if(hlp) *hlp='\0';
20.   strncat(myrname,myname,96);
21.   fprintf(stderr,"Transforming: %s -->",myname);
22.   errors=kc_tag_spec::CSGIOread(stdin, &TheSpec);
23.   if(errors) { fprintf(stderr,"\nError reading AST: %s\n",errors); exit(1); }

24.   for(i=0; i<kc_last_uview; i++) if(!strcmp(kc_uviews[i].name,myname)) uv=i;
25.   for(i=0; kc_rview_names[i]; i++)
26.   if(!strcmp(kc_rview_names[i],myrname)) rv=static_cast<rview>(i);

27.   if(!rv && !uv) { fprintf(stderr," no action defined\n"); exit(2); }

28.   while(rv)
29.   { fprintf(stderr," rewriting(%d)...",cnt); TheSpec=TheSpec->rewrite(rv);
30.     rv=base_rview; sprintf(myrname,"r%d_%s",++cnt,myname);
31.     for(i=0; kc_rview_names[i]; i++)
32.     if(!strcmp(kc_rview_names[i],myrname)) rv=static_cast<rview>(i);
33.   }

34.   if(uv)
35.   { fprintf(stderr," unparsing...\n"); TheSpec->unparse(printer, *kc_uviews[uv].view); }
36.   else
37.   { fprintf(stderr,"\n"); errors=TheSpec->CSGIOwrite(stdout);
38.     if(errors) { fprintf(stderr,"\nError writing AST: %s\n",errors); exit(3); }
39.   }

40.   return 0;
41. } /* main */
```

The back-end program performs the following steps.

1) Find out my own base name and a corresponding rewrite view name (lines 16-20).
2) Read in the BNF abstract tree (lines 22-23).
3) Try to find a matching unparse or rewrite view (lines 24-26).
4) Rewriting: rewrite with all matching rewrite views (lines 28-33).
5) If a matching unparse view is found, then use it for unparsing (lines 34-35).
6) If no matching unparse view is found, then output the BNF abstract tree (lines 36-39).

5.3.2 Common Parts of the Syntax

All of the RSDL lexis and syntax descriptions are given in a more or less modified BNF. So it makes sense to have the same internal representation for all of them. The BNF abstract syntax structure is therefore a modified version of the structure in Section 2.3.1. In contrast to the previous definition in Section 2.3.1, we need here an additional representation including the symbol table.

```
spec: PlainSpec( syntax ) | Spec( syntax symtab );
```

There are two more differences to the description in Section 2.3.1, namely the introduction of a rule type (whether the rule is with ::= or :: or =) and two new atoms: one for prefixed non-terminals (e.g. <agent name>) and one for non-terminal sets as appearing in the abstract syntax.

```
syntax: list rule;

rule:
    Rule( ruletype casestring expression )
  | Token( casestring )
  ;

ruletype: Equality() | Composite() | Unknown() ;

expression: list serial;

serial: list atom;

atom:
    Terminal( casestring )
  | Nonterminal( casestring )
  | PrefixedNT( casestring casestring )
  | AnyAtom( atom )
  | SetAtom( atom )
  | NonZeroAtom( atom )
  | ZeroOneExpression( expression )
  | SubExpression( expression )
  ;

/* symbol table */
symtab: list symbol;

symbol {uniq}:
    NT( casestring )
  | TT( casestring )
  ;
```

After looking at the abstract syntax, we turn to the BNF grammar. Again, this grammar is reused for all RSDL BNF parts. After declaration of auxiliary functions (not shown here) the tokens and the types of the non-terminal nodes are declared.

```
%token                 ASSIGN EQASSIGN COASSIGN SET LEAF
%token <yt_casestring> DEFNT NONTERMINAL NTPREFIX TERMINAL TOKEN

%type  <yt_spec>         spec
%type  <yt_syntax>       syntax
%type  <yt_rule>         rule token
%type  <yt_ruletype>     assigntype
%type  <yt_expression>   expression
%type  <yt_serial>       serial
%type  <yt_atom>         atom
%%
```

The BNF grammar itself is formulated straightforward as already presented in Section 2.3.2.

```
spec: syntax
      { TheSpec = $$ = PlainSpec( $1 ); }

syntax:  /* empty */
      { $$ = Nilsyntax(); }
    | syntax rule
      { $$ = Conssyntax( $2, $1 ); }
    | syntax token
      { $$ = Conssyntax( $2, $1 ); }
    ;
```

```
rule:
      DEFNT assigntype expression
      { $$ = Rule( $2, $1, $3 ); }
    | DEFNT assigntype LEAF
      { $$ = Rule( $2, $1, Nilexpression() ); }
    ;

assigntype:
      ASSIGN
      { $$ = Unknown(); }
    | COASSIGN
      { $$ = Composite(); }
    | EQASSIGN
      { $$ = Equality(); }
    ;

token:
      TOKEN
      { $$ = Token( $1 ); }
    ;

expression:
      /* empty */
      { $$ = Consexpression(Nilserial(), Nilexpression()); }
    | serial
      { $$ = Consexpression( $1, Nilexpression() ); }
    | expression '|' serial
      { $$ = Consexpression( $3, $1 ); }
    | expression '|'
      { $$ = Consexpression( Nilserial(), $1 ); }
    ;

serial:
      atom
      { $$ = Consserial( $1, Nilserial() ); }
    | atom serial
      { $$ = Consserial( $1, $2 ); }
    ;

atom:
      TERMINAL
      { $$ = Terminal( $1 ); }
    | NTPREFIX NONTERMINAL
      { $$ = PrefixedNT( $1, $2 ); }
    | NONTERMINAL
      { $$ = Nonterminal( $1 ); }
    | '{' expression '}'
      { $$ = SubExpression( $2 ); }
    | '[' expression ']'
      { $$ = ZeroOneExpression( $2 ); }
    | atom '*'
      { $$ = AnyAtom( $1 ); }
    | atom '+'
      { $$ = NonZeroAtom( $1 ); }
    | atom SET
      { $$ = SetAtom( $1 ); }
    ;
```

The difference in the BNF languages now comes solely from the lexis. The following table lists the main lexical differences of the various formats.

file names	assign symbol	tokens	non-terminal format
rsdl-lexic.txt	::=	ASCII characters	<a non terminal>
rsdl-cs.txt rsdl-cs-extr.txt	::=	some non-terminals from the lexis and keywords	<a non terminal>
sem-as0.txt	:: and =	some keywords, Token, Int, some non-terminals from the lexis	<a non terminal>
rsdl-as1.txt sem-as1.txt	:: and =	some keywords, (), Token, Int	ANonTerminal

In order to allow a unified handling, the assign symbols are transformed such that all of them start with `::=`, i.e. the two special kinds `::` and `=` are transformed to `::=(::)` and `::=(=)`, respectively.
As an example for formulating the lexis, we will consider the BNF lexis for the abstract syntax BNF below.

```
%{
#include <stdio.h>

char *lexkind = "AS1";

#ifdef DEBUG

  main()
  { char *p;
    printf("checking lexis\n");
    while(p=(char*)yylex())
      printf("%-20.20s is <%s>\n", p, yytext);
  }

  yywrap() { return 1; }

#define MakeCASE
#define MakeLEAF

#define token(x) (int) #x

#else

#include "k.h"
#include "syntax.h"

#define token(x) x

#define MakeCASE { yylval.yt_casestring = mkcasestring(yytext); }
#define MakeLEAF { yylval.yt_casestring = mkcasestring("()"); }

#endif DEBUG

static int yflineno = 1;

void yyerror ( s ) char *s;
{ fprintf( stderr, "syntax error at line %d: %s\n", yflineno, s ); }

%}

NTNAME  [A-Z][A-Za-z\-]*
TERMNAME    [A-Z]+
SPACE   [\t \r]
FULLSPACE   [\t \n\r]

%x comment
%x tokendef
%%

\/\*            { BEGIN(comment); }
<comment>\*\/   { BEGIN(0); }
<comment>.      ; /* ignore characters in comments */
<comment><<EOF>> { yyerror("EOF in comment"); exit(1); }
<comment>\n     { yflineno++; }
::=\(::\)       { return token(COASSIGN); }
::=\(=\)        { return token(EQASSIGN); }
::=             { return token(ASSIGN); }
{TERMNAME}      { MakeCASE; return token(TERMINAL); }
\n              { yflineno++; }
{SPACE}+        { /* ignored */ }
token\(         { BEGIN(tokendef); }
<tokendef>[^)]*  { MakeCASE; return token(TOKEN); }
<tokendef>\)    { BEGIN(0); }
"-set"          { return token(SET); }
{NTNAME}"-set"  { yyless(yyleng-4); MakeCASE; return token(NONTERMINAL); }
{NTNAME}        { MakeCASE; return token(NONTERMINAL); }
{NTNAME}/{FULLSPACE}*:: { MakeCASE; return token(DEFNT); }
\([ \t]*\)      { return token(LEAF); }
\(              { return token('{'); }
\)              { return token('}'); }
.               { return token(yytext[0]); }
%%
```

Please note the use of start conditions here. There is one use for start conditions for tokens. The character sequence `token(tok)` defines a token `tok`, or whatever is inside the parentheses. There are two possibilities to handle such situations. The first possibility is to state the whole lexis including the "`token(`" and "`)`" and then extract the relevant part of the sequence to return as the value of the token, i.e. the characters within the

parentheses. The second possibility is to start a new condition when the open parenthesis appears and to return the characters within the parentheses under this new condition. The new condition ends when "`)`" appears. In the specification above we used the second possibility.
The BNF lexical structure of the concrete syntax BNF and for the lexis BNF are given similarly. There is one peculiarity in the lexis BNF lexical structure. As the RSDL BNF descriptions are checked later on if all terminals used are really declared and if all non terminals used are really defined with a rule, we need to state what should be the terminals of the lexical rules. These are clearly all the ASCII characters, starting from space (code 32) to tilde (code 254). The delete sign is not considered a real sign and all the special signs with codes below space are handled as if they were spaces which is defined in the pre-lexical part.
The definition of the tokens for all characters is given using a special lexical condition `gen_tokens` that is enabled initially (with `YY_USER_INIT`). Within the condition `gen_tokens` always the next character is delivered as token until tilde is reached. Then the condition is left. This special handling is shown below.

```
%{
...

static char tokennum = ' ';

#define YY_USER_INIT { BEGIN(gen_tokens); }
%}

...

%x gen_tokens
%%

<gen_tokens>. {
    unput(yytext[0]);
    if(tokennum=='~') BEGIN(0);
    MakeCHAR(tokennum); tokennum++; return token(TOKEN);
}
...
```

This completes the front-end part of the BNF tooling. The generated abstract representation is stored and the back-end processing can start. There are different kinds of post-processing depending on the input BNF.

5.3.3 Lexis

The backend handling of the lexis BNF has only two parts, namely the generation of a `lex` file and the generation of a token file including all tokens as provided by the lexis BNF. These tokens are used later within the concrete syntax BNF.

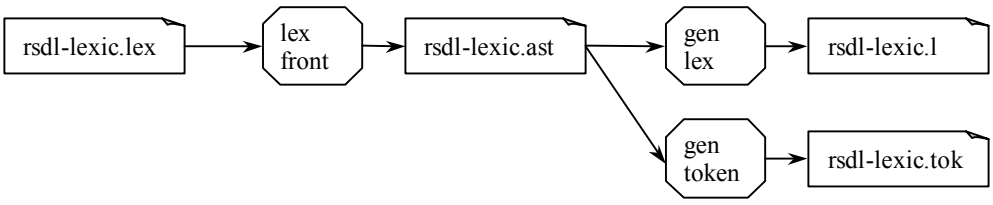


Figure 11: Structure of the Lexical Implementation

The above structure of the lexis BNF implementation is represented by the following `make` rules.

```
${RSDL_L}.lex: ${INPUTS}/${RSDL_L}.txt; ln -f $< $@
%.ast: %.lex lex2ast${EXE}; ./lex2ast${EXE} < $< > $@ || (rm $@; exit 1)
%.l: %.ast ast2l${EXE}; ./ast2l${EXE} < $< > $@ || (rm $@; exit 1)
%.tok: %.ast ast2tok${EXE}; ./ast2tok${EXE} < $< > $@ || (rm $@; exit 1)
```

The first step in the `lex` output generation is the proper generation of cross references using an unparse view `xref`. For every BNF rule the symbol table entry gets an additional attribute `refersto` which holds the reference to its definition. The cross references are used afterwards to navigate from the use of a non-terminal to its definition.

```
%view xref, gentoken, gentokenseq;
symbol {uniq} : { rule refersto = 0; };

r=Rule( name, * ) -> [xref: { symbol sym = NT( name ); sym->refersto = r; } ];

Token( * ) -> [xref: ];
```

5.3.3.1 Token generation

The generation of tokens starts with the generation of the cross references followed by the real token generation.

```
Spec( syn, * )
-> [gentoken: syn:xref syn ];
```

The actual token generation starts from the top level node of the lexical grammar. It is assumed that this is the non terminal which is defined first. Starting from here, tokens are generated according to the following rules.

1. Tokens are generated for the top level node of the lexical grammar.

```
Conssyntax( head, Conssyntax(Token(*), *) )
-> [gentoken: head ];

Conssyntax( *, tail)
-> [gentoken: tail ];
```

2. If the current rule has alternatives with one item each, tokens are generated for the non terminals of all branches.

```
Rule( *, expr=Consexpression( Consserial( *, Nilserial() ), * )),
Rule( *, expr=Consexpression( Consserial( Terminal(*), * ), * ))
-> [gentoken:  expr ];

Consexpression( head, tail )
-> [gentoken: tail head ];

Consserial( head, Nilserial() )
-> [gentoken: head ];

Token( * ),
AnyAtom( * ),
SetAtom( * ),
NonZeroAtom( * ),
ZeroOneExpression( * ),
SubExpression( * )
->[gentoken: ];

Nonterminal( name ),
PrefixedNT( *, name )
->[gentoken: ${ { symbol s=NT(name); } (symbol)s $} ];

s=NT( * )
->[gentoken: s->refersto ];
```

3. A non terminal that has only one alternative is a token.

```
Rule( name, Consexpression( *, Nilexpression() ))
-> [gentoken:  name "::=\n" ];
```

4. A non terminal that is defined to be just one terminal is also a token.

```
Rule( name, Consexpression( Consserial( Terminal(*), Nilserial() ), * ))
-> [gentoken:  name "::=\n" ];
```

5. A non terminal that in all alternatives has only sequences of terminals defines one token for each sequence of terminals. Note: These are the keywords. In fact, they represent two tokens each: the uppercase keyword and the lowercase keyword.

```
c=Consserial( Terminal(t), * )
-> [gentoken: "token(" c:gentokenseq ")\n"];
Consserial( h, t)
-> [gentokenseq: t h ];
```

6. Any other non terminal is itself a token.

```
Rule( name, * )
-> [gentoken:  name "::=\n" ];
```

7. It is an error, when this process reaches a terminal symbol.

```
Terminal(t)
->[gentoken: "error(" t ")\n" ];
```

5.3.3.2 Lex File Generation

The `lex` file generation involves several steps, namely `lex` macro generation for all lexical rules, keyword rules generation and finally lex token rules generation for all tokens as defined by the procedure from the previous chapter. The `lex` file generation starts with the generation of a predefined part.

```
%view ast2l, lexdefs, kwdefs, upper, lower;

Spec( syn, * )
-> [ast2l: "%{\n" "#include <stdio.h>\n#include <string.h>\n\n"
      "extern int yflineno;\n"
      "extern int preyylex();\n"
      "extern int yyparse();\n"
      "extern void yyerror();\n\n"
      "#ifdef DEBUG\n"
      "#define token(x) (int) #x\n"
      "#else\n"
      "#include \"k.h\"\n"
      "#include \"rsdl-cs.h\"\n"
      "#define token(x) x\n"
      "#define YY_USER_ACTION "
      "{ yylval.yt_AS0_rule=AS0_TOKEN(mkcasestring(yytext)); }\n"
      "#endif DEBUG\n\n"
      "#define YY_INPUT(buf,result,max_size) \\\n"
      "  { int c=preyylex(); \\\n"
      "    result = (c==EOF)?YY_NULL:(buf[0]=c, 1); \\\n"
      "  }\n\n"
      "%}\n\n"
```

The next step is the actual generation. It starts with a cross reference generation.

```
      syn:xref syn:lexdefs "\n%%\n\n{NOTE}\t;\n" syn:kwdefs syn
```

The footer is again predefined.

```
      "{SPACE}\t;\n"
      ".\t{ yyerror(\"invalid character\"); }\n"
      "\n%%\n\n"
      "#ifdef DEBUG\n\n"
      "  int main()\n" "  { char *p;\n"
      "    printf(\"checking lexis\\n\");\n"
      "    while((p=(char*)yylex()))\n"
      "      printf(\"%-20.20s on line %4d is <%s>\\n\", p, yflineno, yytext);\n"
      "    return 0;\n"
      "  }\n\n"
"#endif DEBUG\n"
      ];
```

The generation of `lex` definitions from the lexis BNF is straightforward. It is merely another output format for the lexis BNF, namely regular expressions.

```
Conssyntax(h,t)
-> [lexdefs: t h ];

Rule(*,name,e=Consexpression( *, Nilexpression() ))
-> [lexdefs:  name:ucname "\t" e "\n" ];

Rule(*,name,e=Consexpression(Consserial(*, Nilserial()),*))
-> [lexdefs:  name:ucname "\t" e "\n" ];

Rule(*,name,e=Consexpression(Consserial(Terminal(*),Consserial(Terminal(*),*)),*))
-> [lexdefs:  name:ucname "\t" e "\n" ];

Rule(*,name,Consexpression(Consserial(Terminal(t),Nilserial()),Nilexpression()))
-> [lexdefs:  name:ucname "\t[" t:upper "]\n" ];

Token(*)
-> [lexdefs: ];

Rule(*, name, e)
-> [lexdefs:  name:ucname "\t" e "\n" ];

Consexpression( head, tail )
-> [lexdefs: tail "|" head ];

Consexpression( head, Nilexpression())
-> [lexdefs: head ];
```

```
AnyAtom( a )
-> [lexdefs: a "*" ];

Terminal(name)
-> [lexdefs: name ];

Nonterminal(name)
-> [lexdefs: "{" name:ucname "}" ];

SetAtom(a)
->[lexdefs: "set-error(" a ")" ];

NonZeroAtom(a)
->[lexdefs: a "+" ];

ZeroOneExpression(a)
->[lexdefs: a "?" ];

SubExpression(e)
->[lexdefs: "(" e ")" ];
```

The next step is the generation of `lex` rules for the lexis BNF keywords. For every keyword a rule with the lower case and the upper case variant is generated.

```
/* Handling of the keywords */
Conssyntax( head, Conssyntax(Token(*), *) )
-> [kwdefs: head ];

Conssyntax( *, tail)
-> [kwdefs: tail ];

Rule( *, *, Consexpression( *, Nilexpression() )),
Rule(*,*,Consexpression(Consserial(Terminal(*),Nilserial()),Nilexpression())),
Rule( *, *, * )
-> [kwdefs: ];

Rule( *, *, expr=Consexpression( Consserial( *, Nilserial() ), * )),
Rule( *, *, expr=Consexpression( Consserial( Terminal(*), * ), * ))
-> [kwdefs:  expr ];

Consexpression( head, tail )
-> [kwdefs: tail head ];

Consserial( head, Nilserial() )
-> [kwdefs: head ];

c=Consserial( Terminal(t), * )
-> [kwdefs: { if(islower(t->name[0])) }
             ${ c:lower "|" c:upper "\t{ return token(" c:upper "); }\n" $} ];

Token( * ),
AnyAtom( * ),
SetAtom( * ),
NonZeroAtom( * ),
ZeroOneExpression( * ),
SubExpression( * ),
Terminal(*)
->[kwdefs: ];

Nonterminal( name ),
PrefixedNT( *, name )
->[kwdefs: ${ { symbol s=NT(name); } (symbol)s $} ];

s=NT( * )
->[kwdefs: s->refersto ];
```

The generation of the lex rules follows the same algorithm as the token generation. The difference is in the generated string. For every token `tok` we generate a line `{TOK} { return token(TOK); }`.

```
Conssyntax( head, Conssyntax(Token(*), *) )
-> [ast2l: head ];

Conssyntax( *, tail)
-> [ast2l: tail ];

Rule(*,name,Consexpression(Consserial(Terminal(*),Nilserial()),Nilexpression()))
-> [ast2l: "{" name:ucname "}" "\t{ return token(*yytext); }\n" ];
```

```
Rule( *, *, expr=Consexpression( Consserial( *, Nilserial() ), * ))
-> [ast2l:  expr ];

Rule( *, *, expr=Consexpression( Consserial( Terminal(*), * ), * ))
-> [ast2l:  expr ];

Rule( *, name, * )
-> [ast2l:  "{" name:ucname "}" "\t{ return token(L_" name:cname "); }\n" ];

Consexpression( head, tail )
-> [ast2l: tail head ];

Consserial( head, Nilserial() )
-> [ast2l: head ];

Consserial( Terminal(*), * )
-> [ast2l: ];

Token( * ),
AnyAtom( * ),
SetAtom( * ),
NonZeroAtom( * ),
ZeroOneExpression( * ),
SubExpression( * )
->[ast2l: ];

Nonterminal( name ),
PrefixedNT( *, name )
->[ast2l: ${ { symbol s=NT(name); } (symbol)s $} ];

Terminal(t)
->[ast2l: "error(" t ")\n" ];

s=NT( * )
->[ast2l: s->refersto ];
```

Finally, we inspect the generated `lex` file.
First, there are predefined things: declaration of external functions used and debugging support.

```
%{
#include <stdio.h>
#include <string.h>

extern int yflineno;
extern int preyylex();
extern int yyparse();
extern void yyerror();

#ifdef DEBUG
#define token(x) (int) #x
#else
#include "k.h"
#include "rsdl-cs.h"
#define token(x) x
#define YY_USER_ACTION { yylval.yt_AS0_rule=AS0_TOKEN(mkcasestring(yytext)); }
#endif DEBUG
```

Please note the declaration of the macro `YY_USER_ACTION`, which is called whenever a token is analysed. The code given here generates in this case always an AS0 token with an embedded case string containing the token text.
The next step is to define the connection to the pre-lexical part. This is simply done calling the function `preyylex()` as generated by `lex` from the `prelexic-file`.

```
#define YY_INPUT(buf,result,max_size) \
  { int c=preyylex(); \
    result = (c==EOF)?YY_NULL:(buf[0]=c, 1); \
  }
%}
```

This concludes the predefined part. The next part is formed by the definition of the macros for all of the BNF rules of the lexis BNF. Please note, that there is a rule for each of the BNF rules regardless if they are used or not. Examples for entities that are not used further are `LEXICAL_UNIT` and `KEYWORD`. The first of these is not used because it is just a container for declaring the lexical units and the second one because keywords have a special status and are tokens each. Please note that some of the lines have been too long - they have been cut

using \. They are indented by two spaces on the next line. For brevity, some lines are omitted (indicated by ". . .").

```
LEXICAL_UNIT  {NAME}|{CHARACTER_STRING}|{NOTE}|{COMPOSITE_SPECIAL}|{SPECIAL}|{KEYWORD}
NAME {UNDERLINE}*{WORD}({UNDERLINE}+{WORD})*{UNDERLINE}*|{DECIMAL_DIGIT}+\
  (({FULL_STOP}){DECIMAL_DIGIT}+)*
WORD {ALPHANUMERIC}+
ALPHANUMERIC  {LETTER}|{DECIMAL_DIGIT}
LETTER   {UPPERCASE_LETTER}|{LOWERCASE_LETTER}
UPPERCASE_LETTER  A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
LOWERCASE_LETTER  a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
DECIMAL_DIGIT 0|1|2|3|4|5|6|7|8|9
CHARACTER_STRING  {APOSTROPHE}({GENERAL_TEXT_CHARACTER}|{SPECIAL}|{APOSTROPHE}{APOSTROPHE})*\
  {APOSTROPHE}
NOTE {SOLIDUS}{ASTERISK}{NOTE_TEXT}{ASTERISK}+{SOLIDUS}
NOTE_TEXT     ({GENERAL_TEXT_CHARACTER}|{OTHER_SPECIAL}|{ASTERISK}+\
  {NOT_ASTERISK_OR_SOLIDUS}|{SOLIDUS}|{APOSTROPHE})*
NOT_ASTERISK_OR_SOLIDUS     {GENERAL_TEXT_CHARACTER}|{OTHER_SPECIAL}|{APOSTROPHE}
GENERAL_TEXT_CHARACTER {ALPHANUMERIC}|{OTHER_CHARACTER}|{SPACE}
COMPOSITE_SPECIAL {CONCATENATION_SIGN}|{GREATER_THAN_OR_EQUALS_SIGN}|{IMPLIES_SIGN}\
  |{IS_ASSIGNED_SIGN}|{LESS_THAN_OR_EQUALS_SIGN}|{NOT_EQUALS_SIGN}|{QUALIFIER_BEGIN_SIGN}\
  |{QUALIFIER_END_SIGN}
CONCATENATION_SIGN     {SOLIDUS}{SOLIDUS}
GREATER_THAN_OR_EQUALS_SIGN     {GREATER_THAN_SIGN}{EQUALS_SIGN}
IMPLIES_SIGN  {EQUALS_SIGN}{GREATER_THAN_SIGN}
IS_ASSIGNED_SIGN  {COLON}{EQUALS_SIGN}
LESS_THAN_OR_EQUALS_SIGN    {LESS_THAN_SIGN}{EQUALS_SIGN}
NOT_EQUALS_SIGN    {SOLIDUS}{EQUALS_SIGN}
QUALIFIER_BEGIN_SIGN    {LESS_THAN_SIGN}{LESS_THAN_SIGN}
QUALIFIER_END_SIGN     {GREATER_THAN_SIGN}{GREATER_THAN_SIGN}
SPECIAL  {SOLIDUS}|{ASTERISK}|{OTHER_SPECIAL}
OTHER_SPECIAL {LEFT_PARENTHESIS}|{RIGHT_PARENTHESIS}|{PLUS_SIGN}|{COMMA}|{HYPHEN}\
  |{COLON}|{SEMICOLON}|{LESS_THAN_SIGN}|{EQUALS_SIGN}|{GREATER_THAN_SIGN}
OTHER_CHARACTER    {EXCLAMATION_MARK}|{NUMBER_SIGN}|{FULL_STOP}|{QUOTATION_MARK}|{DOLLAR_SIGN}\
  |{PERCENT_SIGN}|{AMPERSAND}|{QUESTION_MARK}|{COMMERCIAL_AT}|{REVERSE_SOLIDUS}\
  |{CIRCUMFLEX_ACCENT}|{UNDERLINE}|{GRAVE_ACCENT}|{VERTICAL_LINE}|{TILDE}\
  |{LEFT_SQUARE_BRACKET}|{RIGHT_SQUARE_BRACKET}|{LEFT_CURLY_BRACKET}|{RIGHT_CURLY_BRACKET}
EXCLAMATION_MARK  [\!]
QUOTATION_MARK    [\"]
LEFT_PARENTHESIS  [\(]
RIGHT_PARENTHESIS [\)]
ASTERISK [\*]
...
TILDE     [\~]
KEYWORD  active|and|block|channel|connect|connection|create|dcl|decision|else|endblock\
  |endchannel|endconnection|enddecision|endstate|env|export|exported|from|gate|import|in\
  |input|join|mod|nextstate|not|now|offspring|or|out|output|parent|provided|referenced\
  |remote|reset|save|self|sender|set|signal|signalset|start|state|stop|task|timer|to|type\
  |via|with|xor
SPACE     [\ ]

%%
```

This concludes the first part of the `lex` file. The next part contains the declaration what the lexical units are and which value to return. The first element is the lexical deletion of <note>. Whenever note is analysed, it is skipped. Please note that there is another rule for <note> further down stating that the token <note> has to be returned. However, the first rule takes precedence and no token <note> will ever reach the parser. The `flex` tool will generate a warning that the second <note> rule is not reachable.

```
{NOTE}    ;
active|ACTIVE { return token(ACTIVE); }
and|AND   { return token(AND); }
...
xor|XOR   { return token(XOR); }
{NAME}    { return token(L_name); }
{CHARACTER_STRING}     { return token(L_character_string); }
{NOTE}    { return token(L_note); }
{CONCATENATION_SIGN}   { return token(L_concatenation_sign); }
{GREATER_THAN_OR_EQUALS_SIGN}   { return token(L_greater_than_or_equals_sign); }
...
{GREATER_THAN_SIGN}    { return token(*yytext); }
{SPACE}   ;
.     { yyerror("invalid character"); }
%%
```

This concludes the second part of the `lex` file. Please note the last rule saying that any other character not covered by the rules above is regarded to be illegal. The `yyerror` routine is defined within prelexic, because this is the only place where line numbers are known. The input to the generated lexer does already contain only spaces instead of special symbols.
The last part contains some more debugging support.

```
#ifdef DEBUG

  int main()
  { char *p;
    printf("checking lexis\n");
    while((p=(char*)yylex()))
      printf("%-20.20s on line %4d is <%s>\n", p, yflineno, yytext);
    return 0;
  }

#endif DEBUG
```

5.3.4 Concrete Syntax

From the concrete syntax BNF several parts have to be generated. The first and most important part is the `yacc` file that resembles the concrete syntax. `Yacc` will then check the grammar for shift reduce conflicts and generate a parser. The second part is the generation of the abstract syntax level 0 (AS0). This is the interface to the static semantic part, which starts with the AS0 and describes the transformation to the AS1. There is also a description of the AS0 within the formal semantics part, which has to match the description generated from the concrete syntax BNF. The last part to be generated is the `kimwitu` representation of the AS0.
For all of the three parts to be generated some grammar corrections have to be made. Therefore a first step is inserted to implement the common grammar changes. This is achieved with the generation of a so-called `GST` file. Starting from that file, the `yacc` output is generated. The problem in the `yacc` generation is that for generating a valid `yacc` file some more grammar changes have to be done. However, for the insertion of syntax tree generation actions as provided by the `kimwitu` representation of the AS0 the original `GST` structure has to be known. Please find an overview of the concrete syntax implementation in Figure 12 below.

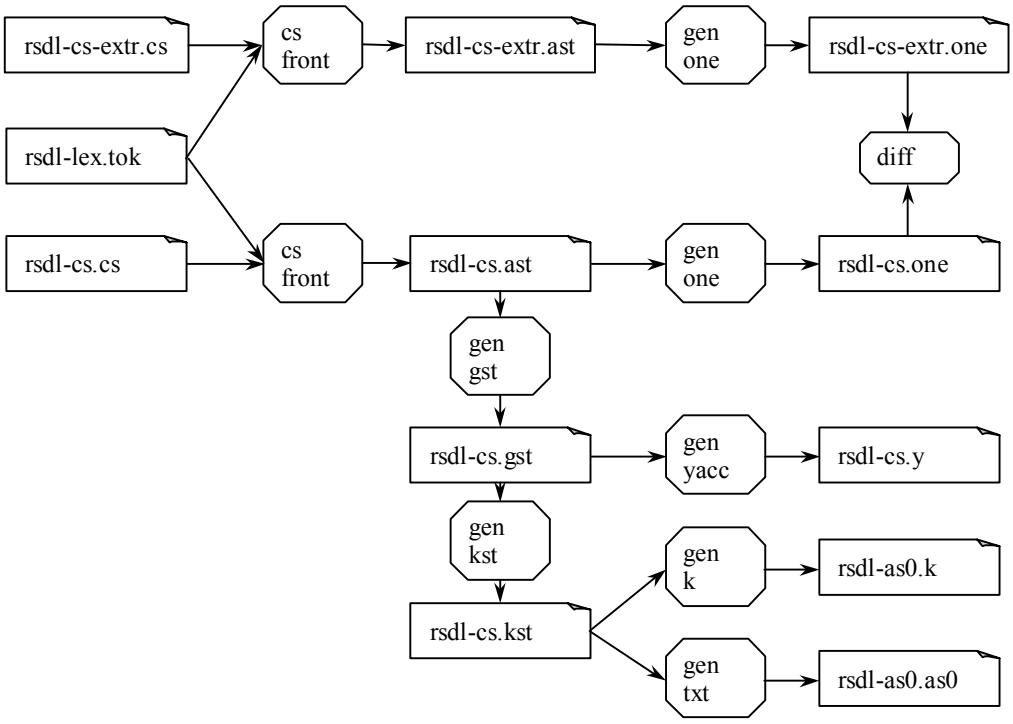


Figure 12: Structure of the Concrete Syntax Implementation

These dependencies are represented by the following `make` statements.

```
${RSDL_C}.cs: ${INPUTS}/${RSDL_C}.txt; ln -f $< $@
${RSDL_O}.cs: ${INPUTS}/${RSDL_O}.txt; ln -f $< $@
%.tok: ${INPUTS}/%.tok; ln -f $< $@
```

```
%.ast: %.cs cs2ast${EXE} ${RSDL_L}.tok
     ( cat $< ${RSDL_L}.tok | ./cs2ast${EXE} > $@ ) || (rm $@; exit 1)
%.gst: %.ast ast2gst${EXE}; ./ast2gst${EXE} < $< > $@ || (rm $@; exit 1)
# gst is a general syntax tree for yacc generation and kimwitu generation
%.kst: %.gst gst2kst${EXE}; ./gst2kst${EXE} < $< > $@ || (rm $@; exit 1)
# kst is a syntax tree for kimwitu generation
%.one: %.ast ast2one${EXE}; (./ast2one${EXE} < $< | sort > $@) || (rm $@; exit 1)
%.y: %.gst gst2y${EXE}; ./gst2y${EXE} < $< > $@ || (rm $@; exit 1)
${RSDL_0}.k: ${RSDL_C}.kst ast2k${EXE}; ./ast2k${EXE} < $< > $@ || (rm $@; exit 1)
${RSDL_0}.as0: ${RSDL_C}.kst kst2txt${EXE}
     ./kst2txt${EXE} < $< | \
     sed -e "s/::/###(::)/" -e "s/=/###(=)/" -e "s/###/::=/" > $@ || (rm $@; exit 1)
cs.diff: ${RSDL_C}.one ${RSDL_O}.one; diff $^ > $@
```

There are some tricky details of the transformation. We will not show all the details of the transformation but only two of the more special parts.

The first detail is the insertion of additional rules. Additional rules are inserted using two steps. First, two new `kimwitu` constructors are introduced (lines 1-2) to trigger the insertion of a new rule (lines 14-20). The insertion itself is done using an auxiliary C-function inserting the new rule into a temporary rule list (lines 6-12). This temporary rule list is inserted into the global rule list when the rewriting reaches the outermost `Spec` node (lines 32-33).

```
1.  atom: MakeRule( casestring atom );
2.  atom: AtomAndRule( atom rule );

3.  %{ KC_REWRITE
4.  static syntax addRules=0;
5.  %}

6.  atom storeRule(a,r) atom a; rule r;
7.  { if(!addRules) addRules=Nilsyntax(); addRules= Conssyntax(r,addRules); return a; }

8.  spec insertRules(s,sy,t) spec s; syntax sy; symtab t;
9.  { syntax loc=(addRules)?concat_syntax(addRules,sy):sy; addRules=Nilsyntax();
10.   if(loc==sy) return s;
11.   fprintf(stderr,"\nadding all rules\n"); return Spec(loc,t);
12. }

13. /* divide up "a b | ..." and "... | a b" into "..." and "a b" */
14. Consexpression(o=Consserial(*,Consserial(*,*)), r=Consexpression(*,*))
15. -> <r_cst2gst: Consexpression(Consserial(MakeRule(NewAS0Name(ser2atom(o)), ser2atom(o)),
16.              Nilserial()), r) >;
17. Consexpression(r=*, Consexpression(o=Consserial(*, Consserial(*,*)), Nilexpression()))
18. -> <r_cst2gst: Consexpression(r, Consexpression(Consserial(MakeRule(
19.              NewAS0Name(ser2atom(o)), ser2atom(o)), Nilserial()), Nilexpression())) >;

20. MakeRule(n,SubExpression(e))
21. -> <: AtomAndRule(Nonterminal(n), Rule(Unknown(), n,e)) >;
22. MakeRule(n,x)
23. -> <: AtomAndRule(Nonterminal(n),
24.             Rule(Unknown(),n,Consexpression(Consserial(x,Nilserial()),
25.               Nilexpression()))) >;
26. MakeRule(*, SubExpression(Consexpression(Consserial(g=GAtom(*,*), *), *)))
27. -> <: g >;

28. AtomAndRule(a,r)
29. -> <: storeRule(a,r) >;

30. s=Spec(sy,t)
31. -> <: insertRules(s,sy,t) >;
```

The second detail is the generation of new names. The problem here is that we would like the name generation to be expressed using unparse-rules, but to call it using a C-function. This is accomplished using a special print function `strprint` as shown below.

```
%{ KC_REWRITE
char buffer[2000] ;

void strprint_f(const char *s, uview_enum v) { strcat(buffer,s); }

casestring NewName(atom a)
{   buffer[0]=0;
    a->unparse(strprint,gen_name);
    return mkcasestring(buffer);
}
```

5.3.4.1 Generation of the AS0 Intermediate Format

For the generation of the intermediate KST format, some more simplifications to the syntax are necessary as already stated above. The transformation of the grammar must be such that the transformed grammar is similar to the original one. Especially it should not be allowed to make too serious transformations. However, the grammar representation for the AS0 does only allow some of the things which are allowed for the concrete syntax. So the following transformations have to be done.

1. An alternative production is mapped to an alias rule.
2. A sequence production is mapped to a constructor rule.
3. Mixed productions are resolved by introducing auxiliary rules.
4. All other productions are mapped to alias rules.
5. Delete <end> or [<end>] everywhere.
6. If one of two lexical units in a row is precious and the other one is not, delete the non precious one.
7. A non terminal followed by a sequence of the same non terminal is merged together.

In order to handle these various special cases appropriately, new node types for precious and non-precious (non)terminals are introduced. These are used later to decide whether to delete an item or not.

```
/* ------------ check out non-precious tokens ----------- */
atom: PreciousNonterminal( casestring );
atom: NonPreciousNonterminal( casestring );
atom: PreciousTerminal( casestring );
atom: NonPreciousTerminal( casestring );

Nonterminal(cs="<name>"),
Nonterminal(cs="<quoted operation name>"),
Nonterminal(cs="<character string>"),
Nonterminal(cs="<hex string>"),
Nonterminal(cs="<bit string>")
-> <r_cst2gst: PreciousNonterminal(cs) >;

Rule(Unknown(),cs="<name>",t),
Rule(Unknown(),cs="<quoted operation name>",t),
Rule(Unknown(),cs="<character string>",t),
Rule(Unknown(),cs="<hex string>",t),
Rule(Unknown(),cs="<bit string>",t)
-> <r_cst2gst: Rule(Composite(),cs,t) >;

Nonterminal(cs="<comment>"),
Nonterminal(cs="<end>")
-> <r_cst2gst: NonPreciousNonterminal(cs) >;

Nonterminal(n) -> <r_cst2gst: PreciousOrNot(n) >;

PrefixedNT(*,n)-> <r_cst2gst: Nonterminal(n) >;

Terminal(n) -> <r_cst2gst: PreciousOrNotT(n) >;
```

5.3.4.2 Generation of the Yacc File

For the generation of the yacc file rsdl-cs.y there are two steps. First the grammar has to be further simplified, as sequences have to be represented by extra rules in yacc. After this simplification two parts of the file have to be generated: the declaration of the tokens and the definition of the rules. Please find below part of the generated yacc file. After a predefined header all tokens as generated by the lexical analysis are declared.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "k.h"
#ifdef __cplusplus
extern "C" {
#endif
int yylex();
void yyerror(char *);
#ifdef __cplusplus
}
#endif
%}

%token <yt_AS0_rule> XOR WITH VIA TYPE TO TIMER TASK STOP STATE START SIGNALSET SIGNAL SET
%token SENDER SELF SAVE RESET REMOTE REFERENCED PROVIDED PARENT OUTPUT OUT OR OFFSPRING NOW
%token NOT NEXTSTATE MOD JOIN INPUT IN IMPORT GATE FROM EXPORTED EXPORT ENV ENDSTATE
%token ENDDECISION ENDCONNECTION ENDCHANNEL ENDBLOCK ELSE DECISION DCL CREATE CONNECTION
```

```
%token CONNECT CHANNEL BLOCK AND ACTIVE
%token '>' '=' '<' ';' ':' '-' ',' '+' ')' '(' '*' '/'
%token L_qualifier_end_sign L_qualifier_begin_sign L_not_equals_sign
%token L_less_than_or_equals_sign L_is_assigned_sign L_implies_sign
%token L_greater_than_or_equals_sign L_concatenation_sign L_note L_character_string L_name
```

Then the node types are declared.

```
%type <yt_AS0_rule> "all rule names"
```

With the line above all non-terminals are declared as being of type `yt_AS0_rule`. Instead of the words "all rule names" there is of course a list of all the rule names generated.
The next part is also predefined, it declares the start symbol and the rule for the start symbol, which is assigning the generated value to the external variable `TheRSDLSpec`. Moreover, the rule for <end> is defined here because <end> is deleted from all rules on the way to the AS0.

```
%start startsymbol

%%

startsymbol:
       rsdl_specification
       { TheRSDLSpec=$1; }
       ;

end:
       semicolon
       { $$=$1; }
     ;
```

The following part is given by the transformation of the concrete syntax BNF rules to `yacc`. We will not insert all of the generated rules here, but only a few in order to show how the transformation is done.
In order to make the generated identifiers more readable we use "bigGenId<number>" instead of long generated identifiers.

```
referenced_definition:
       definition
       { $$=$1; }
     ;

definition:
       agent_definition
       { $$=$1; }
     | agent_type_definition
       { $$=$1; }
     ;

agent_type_definition:
       block_type_definition
       { $$=$1; }
     ;

agent_type_structure:
       valid_input_signal_set bigGenId1 agent_type_body
       { $$=AS0_agent_type_structure( $1, $2, $3 ); }
     | /* empty */ bigGenId1 agent_type_body
       { $$=AS0_agent_type_structure( AS0_UNDEF(), $1, $2 ); }
     | valid_input_signal_set bigGenId1 /* empty */
       { $$=AS0_agent_type_structure( $1, $2, AS0_UNDEF() ); }
     | /* empty */ bigGenId1 /* empty */
       { $$=AS0_agent_type_structure( AS0_UNDEF(), $1, AS0_UNDEF() ); }
     ;

...

asterisk:
       '*'
       { $$=$1; }
     ;
```

For every token, a rule like the one above is generated.

The file is completed by rules for the newly generated identifiers (only one rule shown here).

```
bigGenId1:
       bigGenId1 entity_in_agent
       { $$=AS0_CONS( $2, $1 ); }
     | bigGenId1 channel_definition
       { $$=AS0_CONS( $2, $1 ); }
     | bigGenId1 channel_to_channel_connection
       { $$=AS0_CONS( $2, $1 ); }
     | bigGenId1 gate_in_definition
       { $$=AS0_CONS( $2, $1 ); }
     | bigGenId1 agent_definition
       { $$=AS0_CONS( $2, $1 ); }
     | bigGenId1 agent_reference
       { $$=AS0_CONS( $2, $1 ); }
     | bigGenId1 textual_typebased_agent_definition
       { $$=AS0_CONS( $2, $1 ); }
     | /* empty */
       { $$=AS0_NIL(); }
     ;
%%
```

5.3.4.3 Generation of the AS0 Output

For the generation of the abstract `kimwitu` representation of the AS0 grammar it is only necessary to transform the simplified grammar to a `kimwitu` format. There is one difficulty involved in this process, as `kimwitu` rules are still simpler than the rules for the AS0. Please recall the `kimwitu` rules with their strict distinction between constructors and phyla. However, in AS0 it is possible to have unified rules that put together several other rules, i.e. the constructors thereof. In the end this means that the same constructor might be available for several phyla, which is impossible in `kimwitu`. To overcome this problem we introduce only one phylum for all of the AS0 constructors and define all constructors as constructors of this unified type. With this simplification, it is very easy to generate the `kimwitu` representation of the AS0. However, the typing of the nodes is omitted with this step, as now all nodes have the same type regardless of their defined subtypes. In order to re-introduce type checking for the AS0 we will also generate type checking functions which will be called whenever a new node is constructed. The generated AS0 textual representation does exactly match the one used in Part 4.
Please find below some parts of the generated file `rsdl-as0.k`.
First, the global tree root is declared in the `k.h` header file and the real declaration in the `k.c` file.

```
%{ KC_TYPES_HEADER
extern AS0_rule TheRSDLSpec;
%}
%{
AS0_rule TheRSDLSpec;
%}
```

The second step is to define the abstract syntax tree constructors. For every AS0 constructor rule (with `::`) a corresponding `kimwitu` constructor is defined. We also always define a constructor matching the predefined type *TOKEN*.

```
AS0_rule:
       AS0_TOKEN(casestring) /* predefined Token Constructor */
     | AS0_identifier( AS0_rule AS0_rule )
     | AS0_qualifier( AS0_rule )
     | AS0_path_item( AS0_rule AS0_rule )
     | AS0_sdl_specification( AS0_rule AS0_rule )
     | AS0_agent_type_structure( AS0_rule AS0_rule AS0_rule )
     | AS0_block_type_definition( AS0_rule AS0_rule AS0_rule )
     | AS0_block_type_heading( AS0_rule )
...
```

The last part is again predefined, namely an auxiliary constructor for <end> (which is in fact disposed immediately after creation), a general constructor for missing optional parts (*undefined*) and two list constructors: empty list (`NIL`) and head-tail list construction (`CONS`).

```
     | AS0_end( AS0_rule)
     | AS0_UNDEF()
     | AS0_CONS( AS0_rule AS0_rule)
     | AS0_NIL()
     ;
```

The creation of the textual representation of the AS0 is similar to the pretty printing as defined in Section 2.3.1.2.

5.3.4.4 Abstract Syntax Level 0

For the AS0, only the equality between the generated AS0 and the one extracted out of the semantics must be checked as shown below.

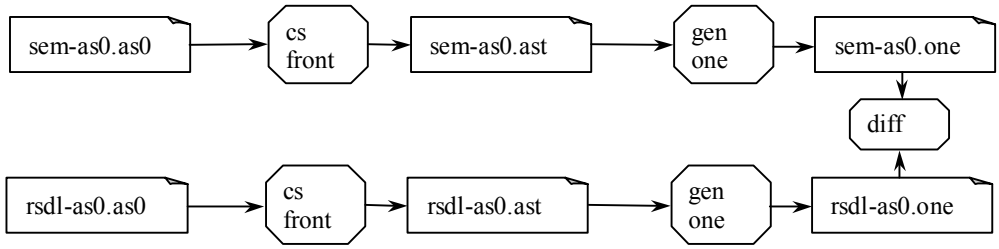


Figure 13: Structure of the AS0 Implementation

This is reflected in the following make rules.

```
${SEM_0}.as0: ${INPUTS}/${SEM_0}.txt; ln -f $< $@
%.ast: %.as0 as0.tok cs2ast${EXE}
     ( cat $< as0.tok | ./cs2ast${EXE} > $@ ) || (rm $@; exit 1)
%.diff: ${RSDL}-%.one ${SEM}-%.one; diff $^ > $@
```

5.3.5 Abstract Syntax

For the abstract syntax AS1 there are two important steps. First the two AS1 representations within the language description and in the semantics have to be equal. Second, a `kimwitu` representation has to be generated. See the following figure for an overview of the steps.

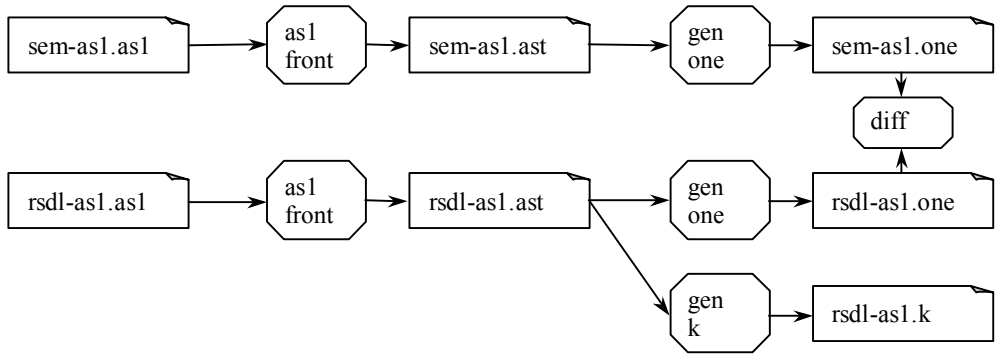


Figure 14: Structure of the AS1 Implementation

The following `make` rules implement the appropriate handling.

```
%.ast: %.as1 as1.tok as12ast${EXE}
     ( cat $< as1.tok | ./as12ast${EXE} > $@ ) || (rm $@; exit 1)
${RSDL_1}.k: ${RSDL_1}.ast ast2k${EXE}; ./ast2k${EXE} < $< > $@ || (rm $@; exit 1)
${RSDL_1}.as1: ${INPUTS}/${RSDL_1}.txt; ln -f $< $@
${SEM_1}.as1: ${INPUTS}/${SEM_1}.txt; ln -f $< $@
```

The abstract syntax has the same format as the AS0 apart from its different structure of non-terminal names. Therefore the same processing can be applied as was already applied for the AS0. Of course, a new general phylum name `AS1_rule` has to be introduced when generating a `kimwitu` output file for the abstract syntax. So please find below part of the `kimwitu` representation of the abstract syntax.

```
AS1_rule:
       AS1_TOKEN(casestring) /* predefined Token Constructor */
     | AS1_Identifier( AS1_rule AS1_rule )
     | AS1_Agent_type_qualifier( AS1_rule )
...
     | AS1_Timer_active_expression( AS1_rule )
     | AS1_end()
     | AS1_UNDEF()
     | AS1_CONS( AS1_rule AS1_rule)
     | AS1_NIL()
     ;
```

5.4 Implementation of the Static Semantics

The static semantics is given by several parts. First, there are transformation rules that transform the RSDL shorthand notations (given in AS0) into their derived counterparts (also in AS1). When doing this, there is also a static analysis included, which means to check whether the identifiers used are really declared, if the types match and the like. This is expressed by well formedness conditions given by Boolean functions that have to be true for the corresponding nodes of the AS0 tree. The transformations are done in several steps. For each step, there are some initial static conditions to check.
The static conditions and the transformation rules are again analysed using the general methodology. They all use a common part of the ASM abstract grammar, namely expressions and patterns as shown below.

```
expr:  Variable(casestring)                 /* a variable name */
    | Literal(casestring)                   /* a literal name */
    | Program(casestring)                   /* a program name */
    | Domain(casestring)                    /* a domain name */
    | IfExpr(expr expr expr)                /* a conditional expression */
    | CaseExpr(expr cases)                  /* a case expression */
    | FunCall(casestring argumentList)      /* a function call */
    | Index(expr expr)                      /* an index expression */
    | MKCall(casestring argumentList)       /* a constructor call */
    | Select(casestring casestring expr)    /* a selection function call */
    | BinOp(casestring expr expr)           /* a binary operator */
    | UnOp(casestring expr)                 /* an unary operator */
    | Quant(qkind nameList expr expr)       /* a quantified expression */
    | SetComp(expr casestring expr expr)    /* a set comprehension */
    | SeqComp(expr casestring expr expr)    /* a sequence comprehension */
    | EmptySet()                            /* an empty set */
    | Undef()                               /* an undefined value */
    ;

cases: list caseentry;                      /* list of case entries */

caseentry: Case(pattern expr defList);      /* one case of a case expression */

qkind:  Exi() | Gen();                      /* quantificator: existential or general */

argumentList: list expr;                    /* list of arguments */

nameList: list casestring;                  /* list of names */

pattern:  NamedPattern(casestring pattern)        /* pattern with a name */
    | ConstructorPattern(casestring parameters)   /* a constructor pattern */
    | MatchAll()                                  /* unspecified pattern ("*") */
    | KeywordP(casestring)                        /* keyword in a pattern */
    | TwoKeywordP(casestring casestring)          /* two keywords in a pattern */
    | UndefP()                                    /* an undefined value in a pattern */
    ;

parameters: list pattern;                   /* list of patterns */
```

These are more than necessary for the transformations, but this way they can also be used for the ASM part. Now we have to insert the rules for the transformations as shown below.

```
definition:  Transformation(letStatements pattern expr casestring expr depTrans);

depTrans: list depTransform;

depTransform:
      DependentTransformation(letStatements expr expr)
    | DependentForall(casestring expr letStatements expr)
    ;
```

The same is true for the conditions, that are now represented by the following additional rule.

```
definition:  Condition(casestring expr);
```

These abstract syntax descriptions are again supplemented by corresponding syntax and lexis descriptions that are omitted here for reasons of brevity. They are really straightforward and do not pose special problems.
The transformation rules are analysed and then transformed into `kimwitu` rewrite rules. In fact, the generated rewrite rules are grouped into different rewrite views in order to reflect the different steps of the static semantics. The conditions are transformed into `kimwitu` functions that are checked using `kimwitu` unparsing rules before and after the individual steps of the static analysis.

5.4.1 Generation of Rewrite Rules from the Transformations

The format of simple transformations does nicely match the `kimwitu` rewrite rules format. It is only necessary to introduce rewrite rule names for the single steps. For simplicity they are named `trans_1`, `trans_2`, etc.
The following special problems have to be handled for the full expressiveness of the transformations:

- auxiliary functions (see Section 5.4.2),
- provided-clauses, and
- multiple rules.

Generation of Provided-clauses
The original `kimwitu` does not allow to formulate conditional rewriting. However, the new version of `kimwitu++` provides support for conditional rewriting as necessary for the RSDL semantics transformations.

Multiple rules
Multiple transformation rules as used within the RSDL semantics description are far beyond the `kimwitu` built-in capabilities. However, they can be implemented using the possibility to state arbitrary C-code within the rewrite rules. This way the tree can be reorganised without `kimwitu` noting it. Utmost care is necessary here in order not to destroy the abstract tree accidentally.

5.4.2 Auxiliary functions

There are three kinds of auxiliary functions, namely derived functions, controlled functions and external or predefined functions. Auxiliary functions are used within the formulation of the transformations and within the static conditions and also within the mapping and in the compilation. Their implementation is given by a corresponding `kimwitu` function.

Derived functions
Derived functions are implemented almost one-to-one in `kimwitu`. The elementary functions used are

- selection functions that are generated together with the abstract syntax structure,
- construction functions that are used as generated by `kimwitu`,
- structural equality which are the `kimwitu` generated `equal` functions

In order to have a better performance for the functions, all auxiliary functions are implemented as memo functions. This means, they are evaluated only once and the result is then stored (memorised) at the corresponding syntax node. A consecutive call of the function with the same arguments is then simply retrieved and not computed once more.

Controlled functions
There is a difference in the handling of derived functions and controlled functions. Derived functions are implemented as memo functions that could access a local node variable for their return value. Controlled functions, on the other hand, are implemented as attributes of nodes with the initial value of *undefined*.

Predefined/External Functions
Predefined ASM functions like concatenation, etc. are implemented as predefined functions over the abstract syntax tree. The functions might include several predefined functions that are available for ASM functions. These predefined functions are declared separately within a file `functions.k`. Especially, there are set and sequence handling functions in this file.
There are also two external function that are heavily used within the transformations and within the compilation, namely the functions *newName* and *uniqueLabel*. The semantics of *newName* is, that it yields for every step a unique new name. However, it is only called once per transformation such that an implementation as C function returning always a new unique name is perfectly valid.[12] The semantics of *uniqueLabel* is to return a new, unique label for each abstract syntax tree node. This is easily implemented using appropriate node attributes.

5.4.3 Generation of a Mapping Function

The mapping from the AS0 to the AS1 is given as a function within the formal semantics description. It will also be implemented as a function in `kimwitu`. The mapping as well as the compilation functions are just very large function definitions. So they are represented as function definitions in the ASM abstract grammar.

[12] Please note the slight difference between the two functions: the correct implementation would have to return the *same* result whenever the function is called twice in the same transformation step.

The mapping function is generated using the function construction means of kimwitu. However, the implementation is particularly easy in this case, as the mapping function is given using patterns that are then transformed to parts that in turn are generated using other patterns. This kind of function definition is very much supported in kimwitu such that the implementation generation is very simple. Please find below a sample part of the mapping as defined within the semantics and the counterpart in the implementation.

```
AS1_rule
Mapping(AS0_rule $a)
{
    AS0_TOKEN(v_x):
         { return AS1_Name(AS1_TOKEN(v_x));  }
    v_i=AS0_identifier(v_q, v_name):
         { return refersto0(v_i)->eq(AS1_TOKEN(mkcasestring("predefined"))) ?
             AS1_Literal(Mapping(v_name)) : AS1_Identifier(Mapping(v_q), Mapping(v_name));  }
    AS0_qualifier(v_q):
         { return Mapping(v_q);  }
    AS0_path_item(AS0_TOKEN("block"), v_n):
         { return AS1_Agent_qualifier(v_n);  }
    AS0_path_item(AS0_TOKEN("block type"), v_n):
         { return AS1_Agent_type_qualifier(v_n);  }
    AS0_rsdl_specification(
      AS0_textual_system_specification_gen_agent_type_definition(v_t, v_s), *):
         { return AS1_RSDL_specification(Mapping(v_t), Mapping(v_s));  }
    AS0_block_type_definition(AS0_block_type_heading(v_name),
      AS0_agent_type_structure(v_entities, v_body), *):
         { return AS1_Agent_type_definition(Mapping(v_name),
         toSet(toSet(Mapping(v_entities))->filter(in_Signal_definition)),
         toSet(toSet(Mapping(v_entities))->filter(in_Timer_definition)),
         toSet(toSet(Mapping(v_entities))->filter(in_Variable_definition)),
         toSet(toSet(Mapping(v_entities))->filter(in_Agent_type_definition)),
         toSet(toSet(Mapping(v_entities))->filter(in_Agent_definition)),
         toSet(toSet(Mapping(v_entities))->filter(in_Gate_definition)),
         toSet(toSet(Mapping(v_entities))->filter(in_Channel_definition)),
         Mapping(v_body));  }
...
    AS0_TOKEN("sender"):
         { return AS1_Sender_expression();  }
    AS0_timer_active_expression(v_id):
         { return AS1_Timer_active_expression(Mapping(v_id));  }
    AS0_UNDEF():
         { return AS1_UNDEF(); }
    AS0_CONS(h,t):
         { return AS1_CONS(Mapping(h),Mapping(t)); }
    AS0_NIL():
         { return AS1_NIL(); }
    default:
         { return AS1_TOKEN(mkcasestring("--error-no match--")); }
}
```

5.4.4 Generation of a Compilation Function

The compilation function is also given in the same way as the mapping function. However, the destination domain, i.e. the domain *BEHAVIOUR* and all the domains below it are not defined within kimwitu. Moreover, the abstract representation of the behaviour is not used within kimwitu but instead has to be transformed to the ASM format. So a simple behaviour domain is defined in kimwitu to be used as the result domain of the compilation. The generated behaviour structure is then output using a simple unparse view as shown below.

```
%uview outputCode, reallyOutputCode;
genCode: list genInstr;
genInstr: GenInstr(casestring genCode);

c=ConsgenCode(*,*) /* wrapper for the predefined part */
-> [outputCode: "transition InitCompilation ==\t\n"
      c:reallyOutputCode "\b" ];

GenInstr(n,NilgenCode())
-> [reallyOutputCode: n ];
GenInstr(n,args)
-> [reallyOutputCode: "mk_" n "(\t" args "\b)" ];
ConsgenCode(h,NilgenCode())
-> [reallyOutputCode: h ];
ConsgenCode(h,t)
-> [reallyOutputCode: t ",\n" h ];
```

5.5 Implementation of the Dynamic Semantics

The dynamic semantics is represented by ASM statements. These statements follow the ASM syntax. They are again analysed using the general methodology. This means, there is a lexis for the ASM part, a grammar and also something to be transformed into. There are several parts of the dynamic semantics. The SAM is plainly represented as ASM and is therefore handled like plain ASM. The same is true for the initialisation modules and the data part.

5.5.1 ASM Abstract Grammar

The remaining parts of the ASM grammar are defined here. The structure of expressions as well as the structure of patterns are already defined within Section 5.4.
An ASM specification is first of all a list of definitions. There are several possible kinds of definitions, namely domain definitions, function definitions, rule definitions and program definitions. Moreover, there could be constraints and initial conditions.

```
asmSpec: Defs(defList);

defList: list definition;

definition:
       DomainDecl(mode casestring)
     | DomainDef(casestring domain)
     | FunctionDecl(mode casestring domain domain)
     | FunctionDef(casestring fargList expr)
     | RuleDef(casestring fargList defList rule)
     | ProgramDef(casestring defList rule)
     | Constraint(expr)
     | InitialCond(expr)
     ;

fargList: list farg;

farg: FArg(casestring domain);
```

For domain declarations and function declarations it is possible to give a mode. Not giving a mode amounts to giving an implicit mode. The modes allowed are declared below.

```
mode: Static() | Controlled() | Shared() | Monitored() | Derived()   ;
```

The next part to define is how domains can be composed. Apart from plain domains we know (power) set domains, list domains, nonempty list domains, tuple domains and union domains. Moreover, a domain could be empty and in some places the domain is not given (e.g. in a function header).

```
domain:    PlainDomain(casestring)
     | SetDomain(casestring)
     | SeqDomain(casestring)
     | SeqPlusDomain(casestring)
     | TupleDomain(tupleDomainList)
     | UnionDomain(unionDomainList)
     | NoDomain()
     | UnknownDomain()
     ;

tupleDomainList: list domain;
unionDomainList: list domain;
```

The next part describes how rules are composed as already explained in Part 2. Possible constructors are assignments, if-then-else constructs, parallel composition (do-in-parallel), for all constructions, choose constructions, extend constructions, let rules and calls of rule macros.

```
rule:      Assign(casestring argumentList expr)
     | IfThenElse(expr rule rule)
     | Empty()
     | Parallel(ruleList)
     | ForAll(casestring expr rule)
     | Choose(casestring expr rule)
     | Extend(domain nameList rule)
     | Let(letStatements rule)
     | RuleCall(casestring argumentList)
     ;

ruleList: list rule;
```

5.5.2 ASM Grammar

The `yacc` description of the ASM grammar is really straightforward and not shown in detail here. However, this is a good place to show the use of precedences. We had the following precedences in Section 2.1.6.

Precedence	Operators
0	$\exists$, $\forall$
1	$\Rightarrow$, $\Leftrightarrow$
2	$\vee$
3	$\wedge$
4	$=$, $\neq$, $>$, $<$, $\geq$, $\leq$, $\subseteq$, $\subset$, $\in$, $\notin$
5	..
6	$+$, $-$, $\cup$, $\setminus$, $\frown$
7	$*$, $/$, $\cap$, **mod**, **rem**, $\times$
8	$\rightarrow$

The are represented by the following `yacc` precedences.

```
%nonassoc ':' /* for quantified */ '|'
%left IMPLIES IFF
%left OR
%left AND
%nonassoc '=' NEQ '>' '<' GEQ LEQ SUBSETEQ SUBSET ELEMENTOF NOTIN
%nonassoc DOTDOT
%left '+' '-' UNION SETMINUS CONCAT
%left '*' '/' INTERSECT MOD REM TIMES
%nonassoc ARROW
%nonassoc UMINUS
```

The expressions themselves need not care about conflicts, because the precedences will solve the shift-reduce-conflicts. Please note the use of the `%prec` for giving a precedence to unary prefix operators in the following partial `yacc` description of ASM expressions.

```
/*************** expressions ***************/
formula:  primary
     | formula '=' formula
       { $$ = BinOp(mkcasestring("="),$1,$3); }
     | formula NEQ formula
       { $$ = BinOp(mkcasestring("!="),$1,$3); }
     | formula AND formula
       { $$ = BinOp(mkcasestring("&"),$1,$3); }
     | formula OR formula
       { $$ = BinOp(mkcasestring("v"),$1,$3); }
     | formula IMPLIES formula
       { $$ = BinOp(mkcasestring("->"),$1,$3); }
     | formula IFF formula
       { $$ = BinOp(mkcasestring("<->"),$1,$3); }
     | NOT formula %prec UMINUS
       { $$ = UnOp(mkcasestring("!"),$2); }
     | IF formula THEN formula elsepart
       { $$ = IfExpr($2,$4,$6); }
     ...
     ;
```

5.5.3 ASM Lexis

For the ASM lexis, the different kinds of identifiers have to be distinguished. In the text this is done using appropriate character styles. However, these styles get lost when generating plain text. Therefore a prefix is generated for the character styles in order to identify them correctly (see also Section 5.2). This is used when lexically analysing the text: after finding such a prefix, `lex` enters a special start condition for this name kind and accepts then the appropriate name. This is exemplified in the following extract of the ASM `lex` file.

```
NAME          [A-Za-z_][A-Za-z0-9_]*
PNAME         [-A-Za-z0-9_]+
SYNNAME       \<[a-z][a-z0-9 ]+\>
...
%x MKN SN KW AN RN DN FN PN
%%
```

```
...
mk-d-/{NAME}  { BEGIN(MKN); }
s-d-/{PNAME}  { BEGIN(SN); }
s-kw-/{NAME}  { BEGIN(SN); }
s-/{SYNNAME}  { BEGIN(SN); }
kw-/{NAME}    { BEGIN(KW); }
d-/{PNAME}    { BEGIN(DN); }
p-/{PNAME}    { BEGIN(PN); }
f-/{PNAME}    { BEGIN(FN); }
r-/{NAME}     { BEGIN(RN); }
a-/{NAME}     { BEGIN(AN); }
...
<MKN>{PNAME}  { BEGIN(0); MakeCASE; return token(MKNAME); }
<SN>{PNAME}   { BEGIN(0); MakeCASE; return token(SNAME); }
<SN>{SYNNAME} { BEGIN(0); MakeCASE; return token(SNAME); }
<KW>{NAME}    { BEGIN(0); MakeCASE; return token(KEYWORD); }
<DN>{PNAME}   { BEGIN(0); MakeCASE; return token(DOMAINNAME); }
<PN>{PNAME}   { BEGIN(0); MakeCASE; return token(PROGRAMNAME); }
<FN>{PNAME}   { BEGIN(0); MakeCASE; return token(FUNCTIONNAME); }
<RN>{NAME}    { BEGIN(0); MakeCASE; return token(RULENAME); }
<AN>{NAME}    { BEGIN(0); MakeCASE; return token(ASMNAME); }
```

5.5.4 Generation of ASM for the ASM workbench

The input format for the ASM workbench is ASM-SL. This is almost the same as the ASM used in Part 4, but there are some problems as already stated in Section 2.3.5.2.

Definition Order
The definition order is implemented using the cross-reference method as already used for the token generation. Every use of a defined entity is linked to its definition. Whenever a new definition should be generated, it is first checked if its constituent parts were already generated. If not, they are generated before. This method is not applicable if there are cycles within the definitions, but this is not the case for the RSDL formal semantics.

Strong Typing
Most of the semantics definitions are already strongly typed. However, especially for the syntax definitions we use heavily union types. Such types are not supported by the workbench. The solution for this problem is the same as the solution for the similar problem in kimwitu: We regard all syntax definitions as belonging to only one type with many constructors. This avoids type-checking in this case.

Predefined Operators and Types
For all predefined operators and domains that are not present in the workbench we generate the corresponding functions in the workbench. Then the mapping of functions to their workbench representation is almost one-to-one.

Agents
It is a strong restriction that the workbench does not support ASM agents. However, there is a way to emulate at least an interleaving variant of agents, which might be enough for test reasons. This emulation is defined as follows.
First we introduce an unspecified domain *AGENT*.

```
freetype Agent == { generator_Agent : INT }
```

The function *Self* is declared to be external (**monitored**).

```
external function Self: Agent
```

This means that whenever this function is to be evaluated, the user is asked about its current value. That way it is possible to choose the agent to be activated next.
The domain *PROGRAM* is defined using identifiers for all the available programs, e.g.

```
freetype Program == { prog1_id, prog2_id, prog3_id }
```

The function *program* is a usual controlled ASM function.

```
dynamic function program: Agent -> Program
```

It remains to compose the main program for all ASM agents as follows.

```
transition MainProgram ==
  if program(Self) = prog1_id then Prog1 endif
  if program(Self) = prog2_id then Prog2 endif
  if program(Self) = prog3_id then Prog3 endif
```

Shared Functions
Shared functions are implemented as special controlled functions. However, a new agent `Env` is introduced with a program modifying the shared function. This way a shared function is implemented making the environment an explicit agent, in accordance with the ASM definition.

Extend
The ASM workbench does not support the extend-Primitive. For single extends this can be mapped to choosing an element that is not yet used. However, there is no way known to emulate extend when it is embedded into a **do forall** loop. This problem is serious. However, for the example we only have single extends. The next version of the workbench is supposed to support extends.

5.6 The Generated RSDL Compiler

The generated compiler consists of the files listed below. Please note that the compiler transforms an RSDL specification into a sequence of ASM statements.

File name	Description
rsdl-lexic.l	Generated `lex` file
rsdl-cs.y	Generated `yacc` file
rsdl-as0.k	Generated file for the `kimwitu` representation of the AS0
rsdl-as1.k	Generated file for the `kimwitu` representation of the AS1
rsdl-fun.k	Generated file for the auxiliary functions
rsdl-trans.k	Generated file for the transformations within the AS0
rsdl-cond0.k	Generated file for the static conditions on the AS0
rsdl-map.k	Generated file for the mapping between AS0 and AS1
rsdl-cond1.k	Generated file for the static conditions on the AS0
rsdl-compile.k	Generated file for the compilation function
rsdl-gen-asm.k	Generated file for the generation of an ASM structure of the syntax tree

The generated RSDL compiler follows again the general methodology. So we have a `lex`-file, a `yacc`-file and a set of `kimwitu` files. Moreover, there are some predefined parts as listed below that are not generated from the language description or the formal definition.

File name	Description
Makefile	Overall description of the file dependencies.
rsdl-prelexic.l	A `lex`-file for the implementation of the special RSDL rules that are before the actual lexis definition. These are the rules for the handling of underlines and for the handling of special characters.
functions.k	Handcrafted file containing ASM predefined functions for `kimwitu`.
rsdlcomp.c	This is the main program of the compiler. It is merely a call of the generated unparse and rewrite functions.

The `makefile` is very simple for this file structure. It follows the general methodology and is therefore similar to all the other `makefiles`.
The first part of the file `rsdl-prelexic.l` is again devoted to the definition of auxiliary functions. In this case, we will have to count the new line characters because all special symbols are transformed into spaces here. However, we want to keep the line information in order to have better error diagnosis.

```
%{
#include <stdio.h>
#define COUNTNEWLINES int i; for(i=0;yytext[i];i++) if(yytext[i]=='\n') yflineno++
int yflineno=1;
%}
```

The second part defines what special symbols are, what an underline is and what a space is.

```
SPECIAL0 [\00\01\02\03\04\05\06\07]
SPECIAL1 [\10\11\12\13\14\15\16\17]
SPECIAL2 [\20\21\22\23\24\25\26\27]
SPECIAL3 [\30\31\32\33\34\35\36\37]
SPECIAL {SPECIAL0}|{SPECIAL1}|{SPECIAL2}|{SPECIAL3}
UNDERLINE "_"
SPACE " "
SPACE_LIKE ({SPECIAL}|{SPACE})
```

This part implements the following RSDL language description:

> The characters in <lexical unit>s and in <note>s as well as the character <space> and control characters are defined by the International Reference Version of the International Reference Alphabet (Recommendation T.50), which is basically the same as ASCII. The lexical unit <space> represents the T.50 SPACE character (acronym SP), which (for obvious reasons) cannot be shown.

As a next part we instruct `lex` to generate another name prefix because we use two generated `lex` files in the same executable. In our case, we choose the prefix "`preyy`" instead of the standard "`yy`".

```
%option prefix="preyy"
%%
```

The next part includes the handling as defined by the language definition.

```
{UNDERLINE}{SPACE_LIKE}+    { COUNTNEWLINES; }
```

> When an <underline> character is followed by one or more <space>s or control characters, all of these characters (including the <underline>) are ignored, e.g. A_ B denotes the same <name> as AB. This use of <underline> allows <lexical unit>s to be split over more than one line. This rule is applied before any other lexical rule.

```
{SPACE_LIKE}+ { COUNTNEWLINES; return ' '; }
```

> A (non-space) control character may appear where a <space> may appear, and has the same meaning as a <space>.
>
> Any number of <space>s may be inserted before or after any <lexical unit>. Inserted <spaces> or <note>s have no syntactic relevance, but sometimes a <space> or <note> is needed to separate one <lexical unit> from another.

```
<<EOF>>  { return EOF; }
.    { return *yytext; }
```

Any other character is not touched by the pre-lexical analysis. The remaining standard footer of the `lex` file is not shown here.

The file `rsdlcomp.c` is built as a combination of the front-end and back-end files of the `Syntax` directory. First, an abstract syntax tree is generated using `yacc` and `lex`. Then this tree is analysed and transformed using the rewrite views `trans_`<number> for the transformation steps and the unparse views `check0_`<number> for the AS0 conditions. After each condition checking step. a global variable is consulted about whether or not the check was successful. The next step is calling the mapping function and the AS1 conditions. Afterwards, the transformation of the tree to ASM format is done using an unparse view followed by the compilation that is implemented using a function and an unparse view. This completes the compiler.

5.7 RSDL Runtime System

The runtime system for RSDL consists of the files listed below.

File name	Description
asm-wb	The ASM workbench executable
asm-wb.cfg	The configuration file for the ASM workbench
general.asm	Predefined ASM functions as defined in Section 2.1.6.
sem-asm.asm	The ASM workbench representation of the SAM, the initialisation and the data
theSpec.asm	The ASM workbench representation of the AS1 tree of the specification and the result of the compilation function when applied to the specification

All the ASM files are loaded into the ASM workbench and then the specification can be interpreted.

Part 6: MISCELLANEOUS

This last part includes various supplementary information. It includes an annotated bibliography, a list of abbreviations, a glossary, proof obligations for the semantics, an overview of the SDL formal semantics project and various indices.

6.1 Annotated Bibliography

This bibliography presents carefully selected references to other work and for further reading. It includes only key references. Moreover, every reference is characterised in a few words.

6.1.1 ASM

[1] Yuri Gurevich: *Evolving Algebra 1993: Lipari Guide*. In E. Börger, editor, Specification and Validation Methods, pages 9-36. Oxford University Press, 1995.

The Lipari guide is the first complete ASM language description. It exactly defines the basic ASM constructs as given here in Section 2.1. Please note, that the earlier work on Abstract State Machines can be found under the name Evolving Algebras.

[2] Yuri Gurevich: *ASM Guide 97*. CSE Technical Report CSE-TR-336-97, EECS Department, University of Michigan-Ann Arbor, 1997.

The ASM Guide 97 defines the current state of the ASM language. However, this guide is incomplete, and it is sometimes necessary to refer to the Lipari Guide for some constructs. Generally the understanding is that both documents define what ASM are. Usually the ASM formal definition need not be consulted in order to understand ASM models due to their similarity with programming languages.

[3] Y. Gurevich and J. Huggins: *The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions*. In Proceedings of CSL'95 (Computer Science Logic), volume 1092 of *LNCS*, pages 266-290. Springer, 1996.

This paper presents ideas how real time can be used within the ASM framework. It introduces a restricted version of real time and of immediate reactions. In the scope of RSDL, this framework is used and extended to capture also more difficult timing aspects.

[4] The Abstract State Machines Home Page: `http://www.uni-paderborn.de/cs/asm/` and `http://www.eecs.umich.edu/gasm/`.

The ASM home page is the best and up-to-date collection of information about Abstract State Machines. It is definitely recommended to visit this place. It includes links to the commented ASM bibliography as well as links to ASM tools.

6.1.2 Semantics Definitions and Implementations

[5] ITU recommendation Z.120 Annex B: *Algebraic Semantics of Message Sequence Charts*. ITU, 1994.

This document defines the formal semantics of Message Sequence Charts in an algebraic way. It includes the semantics for the basic parts of MSC. However, since then the language MSC has changed and this algebraic semantics (axiomatic style) was not able to cope with the new concepts of the language. Therefore a new semantics is worked out which will be denotational in style.

[6] C. Delgado Kloos and P.T. Breuer, editors: *Formal Semantics of VHDL*. Volume 307 of the Series in Engineering and Computer Science, Kluwer Academic Publishers, Boston, 1995.

This book includes several attempts to formalising the hardware description language VHDL, among them the ASM approach that finally became part of the VHDL language standard. It provides a good overview of current semantics definition methods.

[7] E. Börger and W. Schulte: *A programmer friendly modular definition of the semantics of Java*. In J. Alves-Foss, editor: Formal Syntax and Semantics of Java, volume 1523 of LNCS, Springer, 1998.

This definition of the Java semantics is based on Abstract State Machines. It captures the full language Java and is still understandable. This is possible because the language semantics is defined in a modular way, starting with simple concepts and adding more advanced concepts on top of the simpler ones. This results in

several layers of the Java semantics which can be understood separately thus easing the task of overall understanding.

[8] E. Börger and W. Schulte: *Defining the Java Virtual Machine as platform for provably correct Java compilation*. In L. Brim, J.Gruska, and J. Zlatuska, editors: Proceedings of MFCS'98, volume 1450 of LNCS, Springer, 1998.

This formal definition of the Java virtual machine is much in the spirit of the formal definition of the Java language. This makes it possible to formally prove the correctness of compiling Java to the JVM.

[9] Susan Stepney: *High Integrity Compilation – A case study*. Prentice Hall, 1993.

This book invents two languages, a high level programming language and a low level machine language, and implements a compiler from the programming language to the machine language. This compiler is proven to be correct. In fact, the mapping between the two languages is proven to be correct. Parser related issues as syntactical and lexical analysis are not considered. The semantics definitions and the proofs are presented using the Z language.

6.1.3 SDL Language Reference

[10] The International Telecommunication Union Home Page: `http://www.itu.int/`.

The ITU home page contains information about all available standards and information material. You will notice that the main standards are about telecommunication protocols and other telecommunication issues. The languages SDL and MSC are just intended to aid the formal definition of the other standards. Hence, they must also be standardised. Please note, that ITU standards are usually not for free.

[11] The SDL Forum Society home page: `http://www.sdl-forum.org/`.

The SDL Forum Society maintains and updates the SDL and MSC standards. They are passed to the ITU for standardisation. For SDL Forum members the SDL and MSC language descriptions are available. Moreover, all important information regarding SDL and MSC can be found on the SDL Forum page. This includes information about SDL conferences, tools and language changes.

[12] ITU-T Recommendation Z.100. *Specification and Description Language (SDL)*. International Telecommunication Union (ITU), Geneva, 2000.

The SDL standard is published by the International Telecommunication Union (ITU). A new standard is produced every four years. The current language version (commonly called SDL-2000) is of November 1999. Because it took some time until this standard was officially available, it is dated 2000.

[13] ITU-T Recommendation Z.105. *Combined Use of SDL with ASN.1*. ITU, 1993.

The ITU standard Z.105 defines how the SDL built-in data model can be changed to the ASN.1 data model. This is important because ASN.1 is traditionally used to formalise data structures in the scope of telecommunication. In Z.!05, the ASN.1 data constructors are defined in terms of the SDL built-in data constructors. The RSDL semantics (and hence also the SDL-2000 semantics) is specifically designed to allow exchanging of the data part (as defined in Z.105) due to the use of a data interface. For introducing ASN.1 into RSDL it is only necessary to provide the new definitions of the interface functions. Please note, that there is also a new (yet unpublished) version of Z.105 in alignment with SDL-2000.

[14] Kenneth J. Turner: *Using Formal Description Techniques*. John Wiley & Sons, 1993.

This book provides a good introduction into the specification languages SDL, Estelle and LOTOS. It formalises several examples using these three languages. The daemon game example is used there as an example.

6.1.4 Tool References

[15] The `flex` manual: `http://www.gnu.org/manual/flex-2.5.4/flex.html`.

This is a reference to an online version of the flex manual. Flex is the Gnu version of lex and includes all the features of lex. The tooling for the RSDL reference implementation uses flex instead of lex.

[16] The `bison` manual: `http://www.gnu.org/manual/bison-1.25/bison.html`.

This is a reference to an online version of the bison manual. Bison is the Gnu version of yacc and includes all the features of yacc. The tooling for the RSDL reference implementation uses bison instead of yacc.

[17] John Levine, Tony Maron, and Doug Brown: *lex and yacc – 2nd edition*. O'Reilly & Associates, 1992.

If the lex and yacc manuals are not enough, then this book is recommended as a reference for lex and yacc. It is suited for beginners as well as for advanced programmers.

[18] A backtracking `yacc` variant - `btyacc`: `http://siber.org/btyacc/index.html`.

Btyacc is a yacc variant having possibilities for backtracking. This means, that conflicts are not as serious as in the original yacc, if they are in fact caused by missing look-ahead. Btyacc will try one alternative first and go back (backtracking) to the next alternative in case of failure of the first alternative. This tool is a good choice when a grammar is to be parsed that is unambiguous but nevertheless causes conflicts in yacc. However, if the grammar is ambiguous, also btyacc will not help.

[19] ANTLR: Another Tool for Language Recognition: `http://www.antlr.org/`.

ANTLR is a parser generating tool. In a way it subsumes the functionality of kimwitu, lex and yacc. The lexical structure, the concrete grammar and the abstract tree are described in a unified framework. In contrast to yacc ANTLR is based on LL parsing and is able to handle languages with arbitrary look-ahead (as e.g. SDL). ANTLR parse trees can be processed after construction in a similar way as with kimwitu unparse rules (tree walking). Unfortunately, the ANTLR capabilities of conflict detection are worse than that of yacc. Especially, it is possible to write down alternatives that can never be reached (which would be a conflict in yacc).

[20] M. Tofte: *Compiler Generators*. Springer Verlag, 1990.

This monograph described how denotational semantics can be used to generate compilers. It contains an overview of existing compiler generators. The CERES '83 compiler generator, developed by Neil D. Jones and the author, is described in detail. The CERES system serves as an example of a powerful "bootstrapping" technique by which one can generate compiler generators as well as compilers by considering a compiler generator to be a special kind of compiler. The book is suitable for readers who have some practical experience but not necessarily a theoretical background in semantics.

[21] The `kimwitu` home page: `http://purl.oclc.org/net/kimwitu`.

The kimwitu home page contains links to several versions of kimwitu and to the documentation. Kimwitu is a tool that handles abstract syntax trees. It provides means to create trees, to transform them and to generate output from a tree.

[22] The `kimwitu++` home page: `http://site.informatik.hu-berlin.de/kimwitu++/`.

Kimwitu++ is a kimwitu variant that generates C++ instead of C. It was used within the RSDL semantics implementation project. There are some more extensions in kimwitu++ that proved very helpful for the semantics implementation project.

[23] G. Del Castillo: *The ASM Workbench*. In E. Börger, Y. Gurevich, and U. Glässer, editors, Proceedings of the 1999 ASM User Group Meeting at the FM'99, September 1999.

The ASM workbench is a an open tool environment supporting ASM specifications. The ASM workbench builds on the ASM-SL notation (see also [24]). It provides a type checker and a simulator allowing to execute ASM specifications. For more information you will find a link to the workbench at the ASM home page ([4]).

[24] G. Del Castillo. *ASM-SL, a Specification Language based on Gurevich's Abstract State Machines: Introduction and Tutorial*. Technical report (in preparation), Department of Mathematics and Computer Science, Paderborn University.

The language ASM-SL is a strongly-typed version of the general ASM formalism. The typing system is ML-like. ASM-SL also introduces various predefined domains and operators for them. It is a quite concise language and close to standard mathematical notation making it easily readable and understandable.

[25] The Montages System with the Tools GEM-MEX and XASM: `http://www.first.gmd.de/~ma/gem/`.

The Montages framework was originally developed for the formal specification of realistic programming languages. The tool GEM-MEX is a rapid prototyping tool for this framework. Due to the same domain of application, this work bears significant similarities to the work described within this book. However, the use of an abstract syntax as in the case of RSDL with transformations, conditions and mapping is not supported in GEM-MEX. However, recently the ASM variant which is used within GEM-MEX was offered as a separate tool called XASM. This tool is also a strong candidate for the RSDL semantics implementation.

[26] The RSDL semantics: `http://www.informatik.hu-berlin.de/~prinz/Implementation`.

This page includes the complete implementation of the RSDL semantics as described in this book. Moreover, also the complete implementation of the SDL semantics is provided here, as far as already finished. Please note, that the tooling for SDL and the one for RSDL are similar in many respects.

6.1.5 Alternative SDL Semantics Definition Approaches

[27] ITU Recommendation Z.100, Annex F: *Formal SDL semantics*.

The SDL-92 formal semantics is based on an abstract interpreter for the language SDL. This interpreter is formulated using Meta IV. The runtime system is also formulated using Meta IV together with CCS process communication primitives and an external notion of time (ticks). It was not possible to adapt this formal definition to the new needs of the SDL-2000 language, therefore it was decided to define a new semantics.

[28] Ursula Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL*. Dissertation, Fakultät für Informatik der Technischen Universität München. (in German)

This thesis comes to the conclusion that SDL is just semi-formal due to its inconsistent formal semantics. To make SDL formal, a denotational SDL semantics is given in a framework called Focus. However, the approach deviates in some areas from the intuitive SDL semantics, especially for the time semantics. The functional view of Focus leads to cumbersome constructs in the semantics of state transition graphs. Moreover, the static part of SDL (conditions, transformations, syntax) is not covered. The size of the semantics for basic SDL constructs indicates that it is difficult to extend this approach to full SDL.

[29] J.A. Bergstra and C.A. Middleburg: *Process Algebra Semantics of φSDL*. Technical Report UNU/IIST Report No. 68, The United Nations University, April 1996.

The approach with φSDL is to provide a process algebra for a restricted version of SDL, namely a flat one. Moreover, also the notion of time is restricted with respect to SDL. Although a magnificent mathematical framework is used, it is not possible to capture the complete semantics of SDL. The authors state that only with a dramatically reduced version of SDL it is possible to apply advanced analysis and formal verification techniques. This report is only recommended for persons with sound mathematical background.

[30] J. Fischer and E. Dimitrov: *Verification of SDL protocol specifications using extended Petri Nets*. In Proceedings of the workshop on Petri Nets and Protocols of the 16th International Conference on Application and theory of Petri Nets, Torino, Italy, 1995.

Although this paper is not really an attempt of providing a semantics for SDL, it is quite convincing in its results. The approach is to provide a pragmatic mapping between a slightly reduced SDL and slightly extended Petri Nets. However, the simplifications to SDL are not very serious, such that usual SDL specifications can be handled. Moreover, the extensions to Petri Nets are not that serious, such that it is still possible to apply some of the model checking properties of the basic Petri Nets. All this theoretical work is nicely embedded into an appropriate tool environment called SITE. For the tool set of SITE see also `http://www.informatik.hu-berlin.de/Themen/SITE/`.

[31] U. Glässer. *ASM semantics of SDL: Concepts, methods, tools*. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998), volume 2, pages 271-280, 1998.

This paper is one of the sources of the current SDL semantics project. It introduces Abstract State Machines for the SDL semantics definition. This approach was integrated with the other SDL formalisation approaches and lead to the current SDL semantics initiative.

[32] R. Gotzhein, B. Geppert, F. Rößler, and P. Schaible. *Towards a new formal SDL semantics*. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998), volume 1, pages 55-64, 1998.

This paper forms the second source for the current SDL semantics project. It initially proposed to use transition systems to formalise SDL. The concrete way of transition systems was left open at this stage. However, it was concluded that existing methods and tools shall be used for the definition of the SDL formal semantics.

[33] St. Lau and A. Prinz. *BSDL: The Language -- Version 0.2*. Department of Computer Science, Humboldt University Berlin, August 1995.

BSDL is the third parent of the current SDL semantics project. It was an attempt to define a kind of abstract language (namely BSDL) that should be able to formalise SDL afterwards. The formal semantics of BSDL was given in Object-Z. Compared with the semantics definition attempt shown in this book, BSDL would

match approximately the SAM layer of the semantics. The current attempt is to map the language SDL onto this SAM core, while the BSDL attempt was to lift the BSDL core up to SDL.

[34] Uwe Glässer. *Analysis and Verification of Formal Requirement Specifications in Model-Based Engineering of Concurrent Systems*. Habilitation Paper, Department of Mathematics and Computer Science, Paderborn University.

This paper provides an overview of the SDL-2000 formal semantics definition together with the underlying design philosophy and comparison with other approaches. It is recommended to read this paper before reading the SDL-2000 formal semantics definition.

6.2 Abbreviations and Glossary

6.2.1 Abbreviations

AS0	Abstract Syntax level 0 (abstraction of the concrete syntax)
AS1	Abstract Syntax level 1 (proper abstract syntax)
ASM	Abstract State Machines
AST	Abstract Syntax Tree
ITU	International Telecommunication Union
RSDL	Restricted Specification and Description Language
SAM	Special Abstract Machine (in the context of SDL also: SDL abstract machine)
SDL	Specification and Description Language
SDL-2000	SDL as defined in [12]
T.50	ITU reference number of the characters recommendation
Z.100	ITU reference number of the SDL recommendation

6.2.2 Glossary

agent	The term agent is used to denote a block that contains one or more extended finite state machines.
bison	Bison is a variant of yacc.
block	A block is an agent that contains one or more concurrent blocks and may also contain an extended finite state machine that owns and handles data within the block.
body	A body is a state machine graph of an agent.
channel	A channel is a communication path between agents.
environment	The environment of the system is everything in the surroundings that communicates with the system in an RSDL-like way.
flex	Flex (fast lexical analyser) is a variant of lex.
gate	A gate represents a connection point for communication with an agent type, and when the type is instantiated it determines the connection of the agent instance with other instances.
instance	An instance is an object created when a type is instantiated.
kimwitu	Kimwitu is a tool to generate abstract syntax tree handling, see Section 2.3.1 and [16].
lex	Lex is a tool for the automatic generation of a lexical analyser, see Section 2.3.3 and [15].
make	The make tool automatically handles dependencies of files and programs, see Section 2.3.4.
pid	The term pid is used for data items that are references to agents (process identifiers).
signal	The primary means of communication is by signals that are output by the sending agent and input by the receiving agent.
sort	A sort is a set of data items that have common properties.
state	An extended finite state machine of an agent is in a state if it is waiting for a stimulus.
stimulus	A stimulus is an event that can cause an agent that is in a state to enter a transition.
system	A system is the outermost agent that communicates with the environment.
timer	A timer is an object owned by an agent that causes a timer signal stimulus to occur at a specified time.
transition	A transition is a sequence of actions an agent performs until it enters a state.
type	A type is a definition that can be used for the creation of instances.
value	The term value is used for the items of a sort.
yacc	Yacc (Yet Another Compiler Compiler) is a parser generation tool, see Section 2.3.2 and [21].

6.3 Proof Obligations

This chapter is intended to inspect the correctness of the formal RSDL definition. This is done providing a set of proof obligations that have to be true. Furthermore it is sketched how we can assure ourselves that these are really true.

6.3.1 Correctness of AS0 and AS1

a) Correctness of the AS0 compared to the concrete syntax:
The AS0 of the formal semantics has to represent the concrete grammar of the RSDL language definition. This match is quite complex as the language definition is only a presentation grammar and hence ambiguous. However, the grammar selected for this book is relatively tame. Merging equal alternatives together suffices to make it parsable and then a direct match to the language grammar is possible. See also 4.2 for the differences between the concrete and the unambiguous grammars. Things are more complicated in the case of full SDL.

b) The AS0 must be unambiguous.
For RSDL, this is easily shown, because the `yacc`*-representation is of the AS0 has no conflicts. In the case of full SDL, the remaining conflicts would have to be checked that they are not caused by an ambiguity.*

c) Matching of the AS1 definitions:
The AS1 defined in the formal semantics part should match exactly the abstract grammar presented in the RSDL language description. This match is easily shown using a `diff` *functionality.*

6.3.2 Correctness of the Static Semantics

a) Correctness of the functions: type-correctness, computability
The functions in the static semantics have to be computable. Moreover, they have to be type-consistent. This is not easily shown as all tools used are not able to handle union types within their type-checking. Therefore a special type check is needed. However, the type checking is still statically decidable. Most of the necessary type checking is done automatically by the generated kimwitu sources.

b) Correctness of the static conditions: type-correctness
The static conditions have also to be type-correct with respect to the definition of the syntax tree. Moreover, they must be checked against proper use of constructors. Most of this is done automatically by the generated kimwitu sources.

c) Correctness of the transformations: type-correctness, confluent, terminating
The transformations must be checked against the same conditions as the conditions above. Moreover, they must form a terminating rewrite system. This is easily proven as every transformation replaces a shorthand notation by its extended form. The shorthand is never restored afterwards. Therefore, the transformations must be terminating. Another condition to be checked is if the rules are confluent, i.e. that the result of the transformations does not depend on the order of the transformation steps. This is easily validated when one observes that usually the rules do not interfere with each other The only differences are the rules handling definitions and the rules handling states. A different execution order of the rules would lead to a different order of the items in the sequence. However, the elements of the sequences are transformed into sets anyway during the mapping such that the differences disappear.

d) Correctness of the mapping: type-correctness
The mapping must be type-correct in the same way as the conditions and the transformations. However, one has to note that the types of the sides of a mapping match. This condition is almost completely represented in the generated kimwitu code and will be checked by kimwitu or by the C compiler on the kimwitu generated code.

e) Completeness of the transformations and mappings: all constructors are transformed
The completeness constraint is that after applying all transformations the remaining tree is such that all constructors still present will be handled within the mapping. This is easily validated going through all the constructors (:: rules) and checking, whether there is a transformation rule for this constructor or a mapping rule. It would be an error if there is no rule.

6.3.3 Correctness of the Dynamic Semantics

a) Programs: no conflicts in single rules
There is a constraint that in one program it is not allowed to inconsistently change the value of a location. This is regarded to be a specification fault. However, this can be checked in the SAM parts of the semantics and the data part and in fact such changes do not occur.

b) Functions: type-correct, well-defined
All the functions must be type-correct and statically well-defined. This check is performed by the ASM workbench when the function is transformed into the workbench.

c) Programs: lifeness
The specification of the semantics must be such that whenever a transition is possible according to the SDL semantics, also a similar transition is possible within the semantics. This is guaranteed by mapping SDL steps to ASM steps, such that they directly correspond to each other.

6.3.4 Correctness of the Generated Compiler

a) Syntax and Lexis
The syntax and lexis descriptions used within the compiler must match those of the language definition. This is guaranteed in the following ways. The regular expressions used for the lexis representation are just another format of the grammar. They do not differ in other respects. The yacc format and the BNF format match when the auxiliary non-terminals are inserted inline. The kimwitu representation of the abstract syntax also clearly matches the grammar description in that each :: rule matches one-to-one a kimwitu constructor. This way the generated syntax tree is the same as described with the grammar.

b) Transformations
The transformations are mapped to kimwitu rewrite rules. For simple transformations the formats do exactly match. However, for complex rules some more things are done. However, all these changes lead to the fact that a transformation step is one step in the rewriting.

c) Conditions
The conditions are mapped to kimwitu functions which describe the same as the original functions.

d) Compilation and Mapping
The compilation and the mapping are defined in kimwitu in the same way as in the semantics definition.

e) ASM Parts
For the ASM parts the assumption is that the ASM workbench does exactly implement the ASM behaviour. The additional parts are implemented in a way preserving the ASM semantics.

6.4 Applicability of the Methodology

The methodology as presented in Section 2.2 is in fact not only applicable in the scope of the semantics implementation. Instead, it is a general methodology for rapid prototyping of small programs.
There is much literature about the creation of software in the large, i.e. how to handle large projects with many lines of code. However, in the course of any project there is always some amount of work to be spent for rather small pieces of software. These pieces occur as tools for other areas of the project and are often not even dealt with in the project plans. Usually they are done by some expert of some domain who uses some special language and method to create these tools. They will usually work as expected and nobody sees any problem in this approach. However, projects evolve over time and also the requirements for these small tools change. Now the trouble starts. Sometimes the expert who originally designed the tool is no longer available, sometimes she will not understand her own code and usually it will cost much time to adapt the code.
For the scope sketched above the methodology is appropriate. There are several parts of it that tightly belong together. The methodology will make it easy to design tools for such small tasks and it will make it easy to change them later on very easily. This is achieved by the use of meta-tools taking a high-level description of the problem domain and generating code automatically. The general approach is to consider the small tools as some kind of compilers, that take an input of some defined format and generate output of some other defined format. Usually there will be only one input format and many output formats, but it is also possible to have many input formats and one output format. Central to the methodology is the representation of the input in an abstract syntax tree and the distinction of two kinds of tools, namely front-end tools transforming the input into the abstract representation and back-end tools transforming the abstract representation into the output format.
The examples of the RSDL formal semantics implementation may serve as templates for the application of the methodology.

6.5 SDL – A Language with a Formal Semantics

SDL (Specification and Description Language) is an ITU standardised language for the description of distributed systems. Along with the work on SDL-2000, it has become apparent that a new SDL semantics has to be defined, superseding the former semantics which dates back to SDL-88. In several meetings of the SDL language definition group (ITU-T study group 10, question 6: SG10/Q6), the essential design objectives of such a semantics have been clarified. Among the primary design objectives is intelligibility, which is to be achieved by building on well-known mathematical models and notation, a close correspondence between specification and underlying model, and by a concise and well-structured semantics document. Of similar importance and causally dependent on intelligibility is maintainability, since SDL is an evolving language. Because of the language extensions and modifications of SDL-2000, the semantic model has to be sufficiently rich and flexible.
During the discussions within ITU-T SG10/Q6, it has turned out that a prime design objective is the demand for an executable semantics. This calls for an operational formalism with readily available tool support, and a suitable style for defining the semantic model. Subsequent investigations have shown that Abstract State Machines (ASM) meet this and all other design objectives, and therefore have been chosen as the underlying formalism. Meanwhile, the SDL-2000 formal semantics definition has matured and it is generally accepted at ITU to use this semantics definition as the official formal SDL semantics once it is finished.

6.5.1 The Evolution of SDL

SDL-2000 was approved by ITU-T Study Group 10 in November 1999.
Although SDL today is a language applicable to the specification and implementation of distributed systems in general, it has its origins in telecommunications. The development of SDL arose out of a study of the appropriate way to handle stored program control switching systems raised in the ITU in 1968. The result of this study was to agree in 1972 that languages were needed for specification, programming and human machine interaction. The first, small SDL standard was produced in 1976 as the language for specification. Things changed significantly around 1984 as the first tools were being introduced. The tools forced both users and the designers of SDL to be more formal. This required more work, but the benefits were the identification of errors and the ability to animate models, so “what if” questions could be answered. 1988 saw the introduction of a formal definition for SDL in VDM (alias Meta IV) to underpin the natural language description. There was another significant update to SDL in 1992 by the addition of **type** constructs for an object oriented version of SDL.
Uptake of tools was initially slow even within the industry, because the graphical tools were slow and expensive. The situation today is that SDL tools have high functionality, and a proven track record. The SDL tool market has expanded and changed significantly in the last two years. The reason is, that it has become practical to use SDL for the (semi-) automatic generation of implementations. SDL tools can produce source code in programming languages (usually C/C++) directly from an SDL specification and this code can be linked with a run time system to make products. The generated C++ is treated as an intermediate language in much the same way as compilers treat assembly language. Of course, SDL can still be used in an abstract way with informal text, so that SDL is a broad-spectrum language that can be used from requirements capture to implementation.
These latest trends have pushed SDL in two directions: combining it with object modelling techniques (in particular UML), and improving its use for the automatic generation of implementations. This is reflected by the new language version SDL-2000 which includes now features as:

- Exceptions and exception handling;
- New data model, including object data and direct support of ASN.1 with SDL;
- A unified structuring concept for blocks and processes (agents);
- Composite states and state machine decomposition;
- Interfaces, class symbols as references and associations.

The complete description of SDL-2000 consists of the recommendations Z.100(11/99) for the main text, Z.105(11/99) for ASN.1 in SDL modules, Z.107 for ASN.1 embedded in SDL, and Z.109 for SDL combined with UML. Moreover, there are two annexes to Z.100, namely annex D for the predefined data and annex F for the formal semantics.
Work is not yet completed for SDL-2000. The formal definition and the Z.106 Common Interchange Format (CIF) standard are to be updated. These should be approved at a special SG10 meeting in November 2000.

6.5.2 The SDL-92 Formal Semantics

The style of the former SDL semantics has the typical characteristics of an interpreter-approach, as compared to a compiler-approach. An interpreter usually performs lexical and syntactic analysis functions just as a compiler does. Unlike compilers, interpreters execute a version of the source specification directly, instead of producing executables for some kind of machine. Thus, the source specification is viewed as a data structure rather than a collection of instructions.

The interpreter defining the semantics of SDL-92 is written in Meta IV, and consists of two parts. Starting point is an SDL specification in an abstract syntax called AS0. The translation step from the concrete textual SDL syntax to AS0 is not formally defined, but is derived from the correspondence between names in the two syntaxes.[13] In the first step, the AS0 representation is checked for well-formedness conditions, also referred to as static semantics, and is transformed into a representation in another abstract syntax called AS1 by replacing several SDL language constructs. In the second step, the AS1 representation is interpreted, i.e., used as a data structure that determines the set of computations.
While this interpreter-approach is feasible and appealing in general, there have been three problems in this particular case:

- There was no tool support for executing Meta IV programs, although type checks have been performed.
- The interpreter has not been written in pure Meta IV, but uses CSP-like communication. Although it should be straightforward to define a precise semantics for this variant of Meta IV, this has not been done so far.
- There are several places where SDL systems depend on environmental conditions, such as channel delays or spontaneous transitions. These places are rather hidden in the Meta IV interpreter.

6.5.3 The SDL-2000 Semantics Project

Three different approaches for a formal SDL-2000 definition were developed, namely BSDL [33], an ASM based approach [31] and an approach based on process algebras [32]. It was a great luck that these three approaches could be merged together forming the approach presented in this book. From [31], the underlying formalism ASM was taken, from [32] the concentration on executability, intelligibility, and maintainability and BSDL [33] provided the idea and some of the contents of the SDL abstract machine (SAM).
During the course of merging the three approaches, the former SDL semantics was inspected. The benefits of the interpreter approach were considered and a new approach for the SDL-2000 semantics was developed. The idea is, that in practice processing specifications into an abstract syntax representation is simpler and faster than compiling it into some kind of executable code. On the other hand, compilation into code makes executability easy. There is one well-known technique combining the benefits of interpretation and of compilation, which is the technique of abstract code. With this technique, a specification is transformed into an abstract code representation, which is tailored to the needs of the language and also fairly low-level.
More specifically, the new formal SDL semantics defines the ASM code that an SDL-to-ASM-compiler is supposed to generate, thus enabling the application of compiler techniques. Such a compiler is currently being devised. Due to the nature of ASM and the semantics specification, the compiler generated from the formal semantics is intended to be a *reference compiler*. This means, the compiler is correct, because it reflects the formal semantics, but it is not intended to be particularly effective. Note that once an SDL-to-ASM-compiler is available, it should be feasible to adapt it to another target language by modifying just the code generation phase.

The SDL-2000 formal semantics is being defined by the following persons.

Names	Affiliation	email
Andreas Prinz	DResearch Digital Media Systems GmbH, D- 10319 Berlin, Germany	prinz@dresearch.de
Martin von Löwis	Humboldt-University Berlin, Department of Computer Science, D-12489 Berlin, Germany	loewis@informatik.hu-berlin.de
Michael Piefel	Humboldt-University Berlin, Department of Computer Science, D-12489 Berlin, Germany	piefel@informatik.hu-berlin.de
Reinhard Gotzhein	Computer Networks Group, University of Kaiserslautern, Department of Computer Science, D-67653 Kaiserslautern, Germany	gotzhein@informatik.uni-kl.de
Robert Eschbach	University of Kaiserslautern, Department of Computer Science, D-67653 Kaiserslautern, Germany	eschbach@informatik.uni-kl.de
Uwe Glässer	Heinz Nixdorf Institute, University of Paderborn, Department of Mathematics and Computer Science, D-33102 Paderborn, Germany	glaesser@uni-paderborn.de
Wang Ying, Ai Bo, Zhao Yuhong, Zhang Weilei	Beijing University of Posts and Telecommunications, Beijing, China	wyingr@263.net

[13] However, this transformation is almost one-to-one, such that no formality is needed here.

The SDL formal semantics project was conducted in the same way as the RSDL language description. So it has all the parts that played a role within this book. The following table lists the main contributors in the corresponding slots, although almost everybody contributed also to the other areas.

Part of the Formal Semantics Description	Main Contributors (underlined names from above)
Generation of AS0	Michael, Andreas
Static Semantics: Conditions	Ying
Static Semantics: Transformations	Andreas
Static Semantics: Mapping	Andreas
Dynamic Semantics: SAM	Reinhard, Uwe, Andreas, Robert
Dynamic Semantics: Compilation	Andreas
Dynamic Semantics: Initialisation	Reinhard
Dynamic Semantics: Data	Martin
Implementation: Static Part	Michael, Andreas
Implementation: Dynamic Part	Andreas

6.6 Index

6.6.1 Functions

6.6.2 Domains

6.6.3 Concrete Syntax and AS0 Non-terminals

In the concrete syntax and AS0 index, the non-terminal definitions and uses of the (informal) RSDL language definition are set in italics.

<timer definition item> *67*, 81, 92; defined at ***67***, **92**
<timer definition> *55*, 81, 84, 92, 101, 102, 108; defined at ***67***, **92**
<transition gen transition string> 77, 94, 98, 99, 104; defined at **98**
<transition string> *64*, *67*, 76, 77, 98, 99, 104; defined at ***64***, **98**
<transition> *62*, *63*, *64*, *67*, 76, 77, 95, 97, 98, 103, 104; defined at ***64***, **98**
<type expression> 85; defined at ***56***, **85**
<typebased block heading> *56*, 79, 81, 85, 86; defined at ***56***, **85**
<variable access> *70*, 75, 105, 106; defined at ***71***, **107**
<variable definition> *55*, *61*, *71*, *73*, 81, 84, 87, 88, 93, 107; defined at ***71***, **87**
<variable> *33*, *60*, *61*, *63*, *71*, 93, 97, 99; defined at ***71***, **99**
<variables of sort> *71*, *73*, 81, 87, 88; defined at ***71***, **87**
<via gate> *59*, *60*, 90, 91; defined at ***59***, **90**

6.6.4 AS1 Non-terminals

In the AS1 index, the non-terminal definitions and uses of the (informal) RSDL language definition are set in italics.

Active-expression *69*, *70*, 105; defined at ***69***, **105**
Agent-definition *54*, *55*, *56*, *57*, 82, 84, 85, 86, 89, 101, 122, 129, 131; defined at ***56***, **85**
Agent-identifier *14*, *49*, *65*, 100, 101; defined at ***56***, **78**
Agent-name *53*, *56*, 82, 85; defined at ***53***, **78**
Agent-qualifier *53*, 82; defined at ***53***, **82**
Agent-type-definition *54*, *55*, *56*, 82, 84, 85, 129; defined at ***55***, **84**
Agent-type-identifier *56*, 85, 130; defined at ***55***, **78**
Agent-type-name *53*, *55*, 82, 84; defined at ***53***, **78**
Agent-type-qualifier *53*, 82; defined at ***53***, **82**
Assignment *65*, *66*, *71*, 99, 100, 128, 129; defined at ***71***, **99**
Boolean-expression *63*, 97; defined at ***63***, **97**
Channel-definition *55*, *59*, 84, 85, 89, 90, 131; defined at ***58***, **89**
Channel-name *58*, 89; defined at ***58***, **78**
Channel-path *49*, *58*, *59*, 89, 90, 91, 114, 131, 132; defined at ***58***, **90**
Connector-name *64*, *65*, 80, 95, 97, 103; defined at ***64***, **78**
Constant-expression *66*, *69*, *70*, *71*, 87, 103, 105, 109, 127; defined at ***69***, **105**
Continuous-expression *62*, *63*, 97, 126; defined at ***63***, **97**
Continuous-signal *62*, *63*, 95, 96, 97; defined at ***63***, **97**
Create-request-node *64*, 99, 101, 128, 129; defined at ***65***, **100**
Decision-answer *66*, 103; defined at ***66***, **103**
Decision-node *64*, 98, 104, 127, 129; defined at ***66***, **103**
Decision-question *66*, 103; defined at ***66***, **103**
DefinitionAS0 *29*, 77, 79, 81, 82, 83, 89, 91, 92, 93, 104; defined at ***29***
DefinitionAS1 *29*, 77, 79, 80, 81, 112, 117, 126, 127, 129, 130, 131; defined at ***29***
Destination-gate *49*, *58*, 89, 90, 132; defined at ***58***, **90**
Else-answer *66*, *67*, 103; defined at ***66***, **103**
Expression *63*, *65*, *66*, *67*, *69*, *71*, 81, 97, 99, 100, 101, 103, 106, 107, 111; defined at ***69***, **105**
Free-action *50*, *55*, *64*, *65*, 80, 94, 95, 98, 127, 129; defined at ***64***, **97**
Gate-definition *55*, *56*, 84, 85, 89, 130, 132; defined at ***58***, **89**
Gate-identifier *58*, 90; defined at ***58***, **78**
Gate-name *58*, 89; defined at ***58***, **78**
Graph-node *64*, 98; defined at ***64***, **99**
Identifier *49*, *53*, *55*, *56*, *58*, *60*, *63*, *67*, 78, 80, 82, 96, 109, 128, 129; defined at ***53***, **78**
Imperative-expression *69*, 105; defined at ***72***, **105**
Initial-number *49*, *56*, 85, 86, 130; defined at ***56***, **86**
Input-node *60*, *62*, *63*, 95, 96, 97; defined at ***63***, **96**
In-signal-identifier *58*, 89, 132; defined at ***58***, **89**
Join-node *64*, *65*, 80, 102, 103, 127, 129; defined at ***65***, **103**
Literal *69*, *70*, 78, 81, 82, 105, 106, 111, 128, 129; defined at ***68***, ***69***, **106**
Maximum-number *49*, *56*, 85, 86, 87, 122; defined at ***56***, **86**
Name *48*, *50*, *53*, *58*, *60*, *62*, *64*, *67*, *68*, *69*, *71*, 78, 80, 81, 97, 100, 101, 107, 110, 111, 124; defined at ***53***, **78**
Nextstate-node *64*, 80, 102, 127, 129; defined at ***65***, **102**
Now-expression *72*, 81, 105, 107, 128, 129; defined at ***72***, **107**
Number-of-instances *49*, *56*, *57*, 85, 86, 87, 122, 130; defined at ***56***, **86**
Offspring-expression *72*, 107, 108, 128, 129; defined at ***72***, **107**
Operation-application *69*, 81, 105, 107, 111, 128, 129; defined at ***69***, **106**
Operation-name 69, *70*, 81, 106; defined at ***69***, **78**
Originating-gate *49*, *58*, 89, 90, 132; defined at ***58***, **90**
Output-node *60*, *64*, *65*, *66*, 99, 100, 128, 129; defined at ***65***, **100**
Out-signal-identifier *58*, 89, 132; defined at ***58***, **89**
Parent-expression *72*, 107, 108, 128, 129; defined at ***72***, **107**
Path-item *53*, 78; defined at ***53***, **82**
Pid-expression *72*, 81, 105; defined at ***72***, **107**
Qualifier *53*, 78; defined at ***53***, **78**

6.6.5 Macros

6.6.6 Programs